INVESTIGATION OF EFFECT OF DIFFERENT RUN-TIME DISTRIBUTIONS ON SMARTNET PERFORMANCE

Robert Kyle Armstrong, Jr. – Major, United States Marine Corps
B.S., United States Naval Academy, 1985

This thesis investigates, using in-line simulation, the effect of non-deterministic runtime distributions on the performance of SmartNet's schedule execution using the Opportunistic Load Balancing (OLB) Algorithm, the Limited Best Assignment (LBA) Algorithm, an $O(mn^2)$ Greedy Algorithm, and an $O(mn)$ Greedy Algorithm. SmartNet is a framework for scheduling jobs and machines in a heterogeneous computing environment. Its major strength is its use of both current machine loads and predicted job/machine performance when generating schedules. Schedules are built to meet various Quality of Service requirements using the above algorithms among others. We enhanced SmartNet's simulator so that the runtime distributions could be used for experimentation. The distributions were generated using derivations from our study on NAS Benchmarks. Experiments were run for various categories of job/machine heterogeneity to compare the algorithms which account for both load and expected performance (the Greedy algorithms) against OLB and LBA.

For all categories of heterogeneity, the greedy algorithms outperformed the other two algorithms for both truncated Gaussian and exponential distributions. For these same distributions, the $O(mn)$ Greedy algorithm performed as well as the $O(mn^2)$ Greedy algorithm when the heterogeneity of jobs and machines was high.

Master Of Science In Computer Science

Advisor: Debra Hensgen, Department of Computer Science
Second Reader: Taylor Kidd, Department of Computer Science

Unclassified/A

INVESTIGATION OF EFFECT OF DIFFERENT RUN-TIME DISTRIBUTIONS ON SMARTNET PERFORMANCE

Robert Kyle Armstrong, Jr.
Major, United States Marine Corps
B.S., United States Naval Academy, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1997

Author:

　　　　　　　　　　　Robert Kyle Armstrong, Jr.

Approved by:

　　　　　　　　　　　Debra Hensgen, Thesis Advisor

　　　　　　　　　　　Taylor Kidd, Second Reader

　　　　　　　　　　　Ted Lewis, Chairman, Department of Computer Science

ABSTRACT

This thesis investigates, using in-line simulation, the effect of non-deterministic runtime distributions on the performance of SmartNet's schedule execution using the Opportunistic Load Balancing (OLB) Algorithm, the Limited Best Assignment (LBA) Algorithm, an $O(mn^2)$ Greedy Algorithm, and an $O(mn)$ Greedy Algorithm. SmartNet is a framework for scheduling jobs and machines in a heterogeneous computing environment. Its major strength is its use of both current machine loads and predicted job/machine performance when generating schedules. Schedules are built to meet various Quality of Service requirements using the above algorithms among others. We enhanced SmartNet's simulator so that the runtime distributions could be used for experimentation. The distributions were generated using derivations from our study on NAS Benchmarks. Experiments were run for various categories of job/machine heterogeneity to compare the algorithms which account for both load and expected performance (the Greedy algorithms) against OLB and LBA.

For all categories of heterogeneity, the greedy algorithms outperformed the other two algorithms for both truncated Gaussian and exponential distributions. For these same distributions, the $O(mn)$ Greedy algorithm performed as well as the $O(mn^2)$ Greedy algorithm when the heterogeneity of jobs and machines was high.

TABLE OF CONTENTS

ix

LIST OF FIGURES

LIST OF TABLES

I.   INTRODUCTION

This thesis investigates the effect of non-deterministic run-times on the performance of jobs scheduled by SmartNet [Ref. 1, 2, 3, 4] in a heterogeneous computing environment. It has already been shown that if jobs are scheduled by SmartNet, and they run for exactly the expected amount of time, that the overall performance of the system is improved. SmartNet currently computes the expected run-time of a job by averaging previous run-times which it stores in its database after a job terminates. However, jobs rarely run for exactly this expected amount of time; even if a job is run repeatedly with exactly the same parameters, on exactly the same machine, run-times may differ due to memory stalls. Under less ideal conditions, when a job is using a data file located on a remote file server, run-time variations become even more pronounced. When the value of parameters are changed, the run-time can be drastically different. SmartNet attempts to account for parameter value changes using a concept called "compute characteristics" [Ref. 5], but it will often be the case that, at any given time, at least one job will be running with some unidentified compute characteristics. Therefore, this thesis seeks to identify the expected performance of jobs in computing environments where there are changing or unknown compute characteristics. In particular, it focuses on the time of completion of the last job. It compares SmartNet performance under these conditions against performance without SmartNet. Specifically, it compares some of SmartNet's intelligent algorithms, which use expected run-times, against another scheduling algorithm that does not use expected run-times: Opportunistic Load Balancing (OLB). SmartNet's intelligent algorithms have been shown to outperform this algorithm when jobs do run for exactly their expected run-times; this thesis will document the comparison of SmartNet against this algorithm when the actual run-times of jobs are non-deterministic.

To relate the research in this thesis to other fields, we now present an example that demonstrates how we can convert parameters that are typically random and un-

1

controllable into more predictable and expected factors. The idea is to be able to exercise more control on the input to an algorithm that incorporates multiple parameters, many of which may be environmental factors, so that the unpredictable nature of the algorithm's output is lessened. To some degree, an algorithm can then be made more useful.

A real world example of this situation is that of providing indirect fire. Mortars and artillery are indirect fire weapons. Indirect fire is the delivery of explosive ordnance along a parabolic or near-parabolic path from the weapon to the target. This is different from the way rifles, pistols, and tanks deliver ordnance, which is along a line of sight path from the weapon to the target. The parabolic path of artillery allows ordnance to be delivered across great distances and over significant terrain such as hills. A parabolic path, however, allows more factors, many of them uncertain, to influence the outcome of an indirect round. It is the way that these uncertainties are accounted for that is the crux of our example.

Figure 1 shows how indirect weapon fires might impact against a target. The nature of indirect fire causes impacts near the target to disperse mostly along the gun-target line but also somewhat left and right of that line. The resulting footprint is basically an elliptical pattern with the majority of the impacts lying near the center of the ellipse. This is because rounds fired indirectly are subject to the effects of wind, temperature, and the rotation of the earth. Because velocities of rounds are slower, the time of flight of a round is longer, and it is subject to effects not normally considered by a line of sight weapon system. There are also factors particular to the weapon system that can cause rounds to impact with limited precision, as shown in Figure 1. The temperature of the gun tube, the temperature of the powder used to fire the round from the tube, and the seat of the artillery round against the inside of the tube all effect whether the round is fired optimally. If a round is fired optimally, we expect that round to hit the target. If factors such as tube and powder temperature or the effects of wind at higher elevations are not considered in the solution, we expect the

2
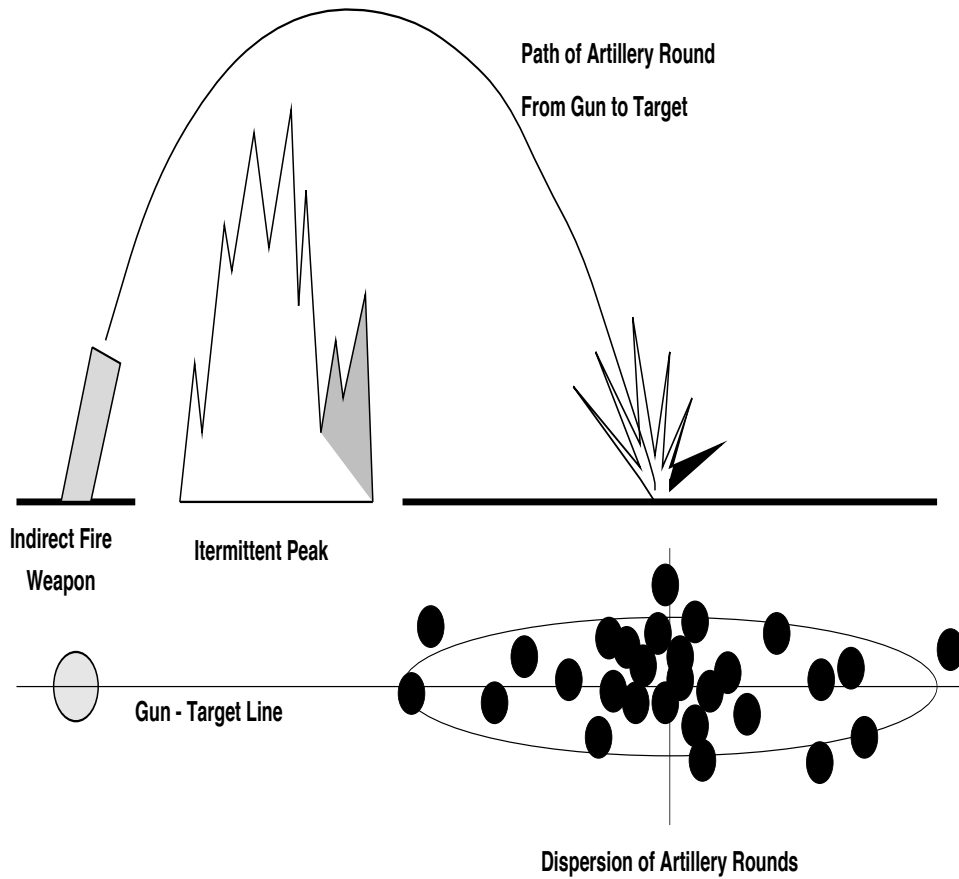
Figure 1. The Random Nature of Artillery Fires.

round may miss.

The artillery community strives to reduce the number of unknown variables present in indirect fire. There are parameters that are external to the artillery mechanism that are major influences upon the outcome. These influences can be measured and their effect compensated for. The artillery community has taken a considerable amount of time and effort to understand, develop tools for measuring, and compensate for these influences. If consistent and timely measurements are made and applied to the artillery solution, we can minimize the affects of outside influences and shoot "first round, on target" with impunity.

It is the reduction of unknowns which is the eventual goal of this thesis. That is, this research strives to understand the external influences upon SmartNet that

might keep it from performing optimally and to determine how best to compensate for these influences. This thesis begins upon this problem by striving to understand the impact of unknowns upon SmartNet's schedules.

A.   BACKGROUND INFORMATION

Scheduling, in general, is a difficult problem [Ref. 6]. As an example, consider the task of scheduling troop and equipment movement from the United States to the east coast of Africa. We describe our example in terms of optimization theory. There are many factors that need to be considered in order to create a schedule for troop and equipment movement. One of the first and most obvious considerations is to determine the maximum possible movement rate of troops and equipment into the area. Only after this maximum movement rate is determined, can scheduling begin. The following additional factors must then be considered:

- The mission commander will set priorities on units and equipment. He will also specify times at which units and equipment must arrive in the theater. The deadlines serve as **scheduling constraints**, whereas the priorities will be incorporated into the **optimization function**.

- Certain pieces of equipment can only be transported by the largest aircraft or by ship. These additional constraints often result in higher transport time.

- An additional example of constraints is the need for a Marine unit to arrive on foreign soil within 72 hours of an identified crisis. The footprint of a forward deployed unit will be small, and their sustainment capability limited to 30 days. Deployment of this unit into the area of operations needs to be planned for; furthermore, the effect of placing a unit into the area of crisis on the will of the foreign force to wage war must be incorporated into the optimization criteria.

- Unfortunately individual threats cannot be considered as **local optimization problems**. We have a large number of air transport assets that are committed globally, which means that 100% of these assets can never be committed to a single local contingency.

- Unfortunately, variables specific to location, such as airfield capacity, may need to be separately modeled throughout the world. Although movement of troops

and equipment by air from the US is very flexible, the movement of troops and equipment into a foreign port or airfield may not be.

- Time is another very important consideration. Time must be managed as effectively and efficiently as possible, and if possible, used to advantage. It takes time to match a contingency plan to the actual scenario, to start the plan, to actually follow the plan, and to revise and correct the plan. The amount of time a commander thinks he has to build up his forces will help him set his priorities for the arrival of equipment and units in theater.

Unfortunately, a single schedule will not suffice. Many "what-if contingencies" need to be calculated; situations can change quickly and schedules must change to accommodate the dynamically changing environment. Being flexible and adaptable are hallmarks of success in any military operation. Constant updates of the current state of movement into the area are required to ensure that the schedule is still valid and effective. Planes and ships and trucks break down, weather changes for the worse, new regional contingencies pop up, and political pressures rise and fall. Schedules must be recalculated to take into account both opportune advantages and unexpected problems. It is the challenge of the scheduler to determine and properly analyze the current state of deployments and movement, as well as the causes of any changes. In summary, we cannot predict exactly how long any given transport operation will require, but we can often match the transport operation mean time and variance to a common probability distribution such as Gaussian or exponential.

The creation of a movement schedule in the above example will also be limited by accurate state information. Acquiring total knowledge of an environment, and a complete understanding the interoperability of the assets in that environment, is a challenging problem. Scheduling decisions are, more often than not, made with limited, and often only "best guess" information. This type of decision making will only reach an optimal solution by accident; a scheduling tool that accounts for variance in transport times would be very useful to commanders in charge of these operations. This thesis will advance the state-of-the-art in heterogeneous schedulers that can, in

the future, not only be applicable to scheduling in computing environments, but also to the problem of scheduling troop movement.

As we have hinted, our example above has direct correlation to a heterogeneous computing environment. In a heterogeneous computing environment, machines of different architectures are often linked together via a network. The machines may be located in the same room or on different continents, or aboard sea-going vessels or on satellites. The variety of architectures in the heterogeneous system provide capabilities above and beyond what you would find in an environment consisting only of machines with similar architecture. Below is an example that illustrates these additional capabilities.

Consider the capabilities of the Single Instruction, Multiple Data (SIMD) machine.

> SIMD machines (Single Instruction, Multiple Data) are an inexpensive way to construct parallel computer systems. A typical SIMD architecture is illustrated in Figure 2.
>
> A single front end controls the entire system; the front end fetches and decodes instructions. It includes (typically) a scalar processor core (usually a RISC machine), plus additional instructions to control the parallel processor ensemble. The front end usually has its own memory to hold the program and scalar data.
>
> The back end comprises many (up to thousands) processing elements (PEs). Each can perform arithmetic operations, memory fetches, and can send and receive messages. The systems essentially replicate the data path of a processor in each PE, but the control part of the processor resides only in the front end. This makes SIMD machines economical to design and build.
>
> When the front end issues a parallel instruction, it broadcasts the instruction to all PEs, which all execute the instruction in parallel. Thus, a single instruction is performed on all data simultaneously. [Ref. 7, pages 746—747]

The capability of a SIMD architecture is maximized, then, when used with programs that require the same instruction or set of instructions be performed on many different "pieces" of data. For example, SIMD machines manipulate matrices better than single processor machines.

Another machine that might be found in a heterogeneous system is a vector processing machine such as a CRAY. CRAY computers set the standard for high performance vector super-computing, and are still utilized worldwide[1] when there is a need for enormous computational capability. The Y-MP EL, a CRAY mini-supercomputer, provides pipelining and segmentation, which are integral features of this architecture that support parallel processing aboard a single chip. Vector processing is provided to enable a programmer to sustain maximal I/O CPU throughput. Vector processing increases computing speed because the execution of single instruction can allow an operation to be performed sequentially on a set (or vector) of operands. [Ref. 8] This type of architecture is suitable for analyzing vectorized data, such as weather or satellite information.

In order to maximize the use of a heterogeneous computing environment consisting of diverse architectures such as the CRAY and SIMD machines discussed above, knowledge of both the machines in the environment and the programs to be run on each machine are required. It may be a waste of compute power to run a job on a machine that is not best suited for the job. Such run-times could be large enough to retard productivity and efficiency even on a lightly loaded system. The problem is compounded on a heavily loaded system. Often, throughput maximization is a goal. Throughput maximization in a heterogeneous environment might mean optimal use of the resources, such that a minimal number of compute cycles are "wasted" doing work better suited to the capabilities of other architectures or machines.

SmartNet is a scheduler that attempts to compute the best scheduling policy for tasks in a shared, heterogeneous computing environment. Such a situation is analogous to the previous troop and equipment movement example. The transport mechanisms are comparable to various machines in a heterogeneous system. Jobs needing to be run on a heterogeneous system are comparable to the units and equipment that need to be moved. The commander needs as much information as possible in order to create

---

[1]CRAY machines are now manufactured by Silicon Graphics, Incorporated.

a near-optimal schedule.

SmartNet is also analogous to the military logistical planner. SmartNet is discussed in detail in Chapter II of this thesis. SmartNet is a scheduling framework for heterogeneous computing environments. It manages both jobs and machine resources in that environment. SmartNet manages these assets by creating a near-optimal schedule of jobs to be run on machines located on the network. SmartNet takes many factors into account, including the performance of jobs on the various architectures, the compute characteristics of a job, current machine loads, and the state of the heterogeneous system. [Ref. 1]

B.   STATEMENT OF PROBLEM

Prior to our research, SmartNet had a rudimentary simulation mode that allowed its scheduling policies to be assessed without tying up the network and wasting valuable compute cycles on machines that may or may not be "owned" by the testing facility. The SmartNet simulator built a schedule from a set of requested jobs and a database containing information about jobs, machines, and job-machine pairs. In simulator mode, SmartNet then performed a discrete event simulation of the execution of the schedule. This previous SmartNet simulator uses the expected time to compute (ETC) value, which is the average run-time of the previous run-times of the job on the same machine, as the simulated run-time. The problem with using ETC values for run-times is that hardly, if ever, will a job execute for exactly the amount of time expected. The use of the ETC value for simulated run-time duration means that the SmartNet simulator does not produce realistic simulation results.

C.   GOAL

The goal of this thesis is to investigate the effects that different run-time distributions have on the performance of SmartNet. We will enhance the SmartNet simulator to provide, as the simulated run-time, a randomly generated run-time from

8

a reasonable run-time distribution for each job. This enhancement will enable us to investigate the efficiency of schedules resulting from the different scheduling algorithms available in SmartNet under more realistic conditions. Simulations using our modified simulator will contribute to an understanding of the value of SmartNet in less controlled environments, such as in the DOD's Joint Task Force Advanced Technology Demonstration (JTF-ATD) and Battlefield Awareness and Data Dissemination (BADD) programs. Additionally, although not part of this thesis work, such schedulers will likely become useful to commanders in the logistical scenario described in our above example.

D.   THESIS ORGANIZATION

This thesis is organized as follows. Chapter II provides a detailed look at SmartNet. Chapter III is concerned with discrete event simulation as it pertains to SmartNet simulation mode. Chapter IV deals with the enhancements that we made to the SmartNet simulator. Chapter V details the experiments performed with the enhanced simulator, as well as the results obtained from these experiments. Chapter VI summarizes the conclusions drawn from these experiments and discusses further related research opportunities.
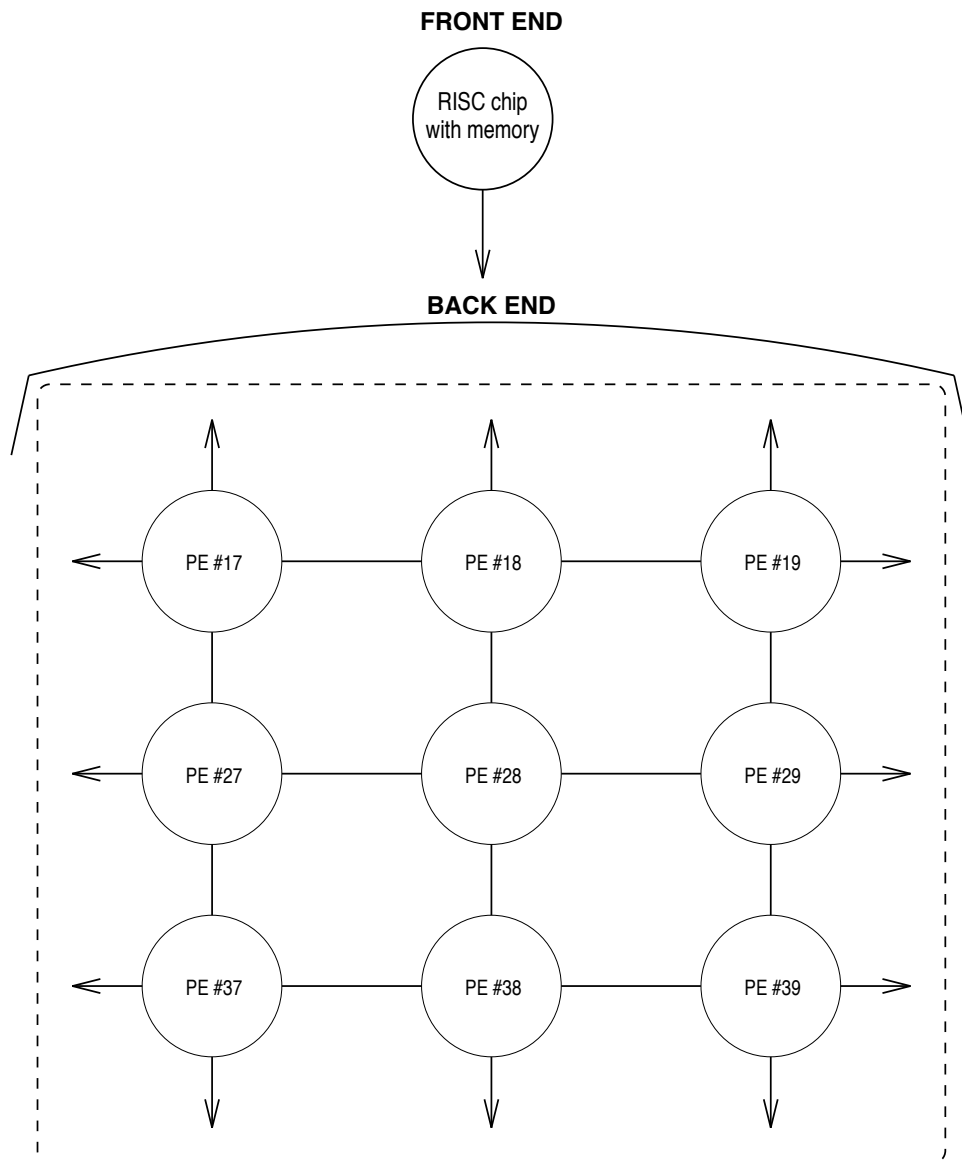
Figure 2. Single Instruction, Multiple Data (SIMD) Machine Architecture. The front end is a RISC chip with memory, used to control the back end. The back end is a matrix arrangement of relatively cheap processors. Each processor performs the same operation on different data, as directed by the front end. In this figure, only a small portion of the back end is shown. Actual matrices of processors can be quite large, up to 32, 64, 128 processors or more.

10

II.   SMARTNET

A.   INTRODUCTION

This chapter describes SmartNet in considerable detail. Section B provides general information about SmartNet and why it was developed. Section C describes how SmartNet operates. Section D contains information about the architecture of SmartNet. Section E summarizes some previous results from the application of SmartNet to scheduling problems. Finally, Section F provides examples of the Opportunistic Load Balancing, Limited Best Assignment, and other SmartNet scheduling algorithms.

B.   BACKGROUND INFORMATION

SmartNet is a framework for scheduling resources in a heterogeneous computing environment [Ref. 1]. It has been in development for over 10 years at the Naval Command, Control, and Ocean Surveillance Center (NCCOSC) Research, Development, Test and Evaluation (RDTE) Division, San Diego, California. The principle scientist is Richard Freund; however, the SmartNet Development Team consists of government employees and contractors working in various locations across the United States. The software currently contains over 100,000 lines of code, developed with 24 staff-years of effort.

The computing world is full of heterogeneous computing environments. They exist wherever machines with distinctly different architectures are networked. The machines may be connected for any number of reasons, but the environment that most demands a product with SmartNet's capabilities is an environment used to perform input-output intensive [Ref. 9] and/or compute intensive jobs [Ref. 1].

Current and future high performance computing (HPC) applications need increasing amounts of computing power. Because of this, there is an increasing focus on maximizing the productivity and efficiency of all available computing assets. In most HPC centers, local and remotely available computers comprise a heterogeneous

11

network. By allowing all of these assets to be utilized by a maximum number of applications, the connected assets in effect become a metacomputer. Figure 3 is a pictorial description of this concept.
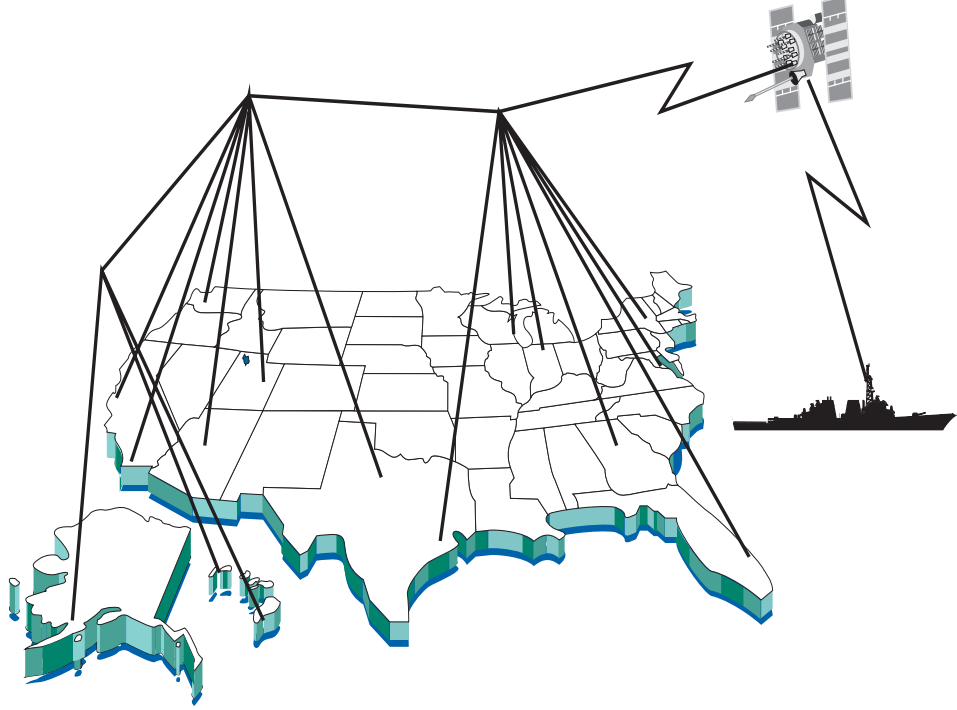
Figure 3. The Metacomputer Concept. Many HPC sites are connected to form a large, powerful, distributed virtual machine.

Ongoing efforts within the research community include creating distributed computing environments (DCEs) in order to further maximize the potential compute power of these heterogeneous assets. Resource management systems (RMSs) have been incorporated into existing computing environments with the goal of better managing the set of resources. DCEs and RMSs have fostered improvements in HPC, but still do not tackle the difficult problem of scheduling jobs and machines intelligently.

SmartNet is capable of supplementing the efforts of DCEs and RMSs to more fully maximize the compute capability in a heterogeneous computing environment. Its

focus is on optimizing a *set* of tasks instead of each task singly. [Ref. 10]

While SmartNet is not the only advanced scheduling system under development, it does have features that distinguish it from other packages. Most scheduling efforts to date utilize Opportunistic Load Balancing (OLB) to develop scheduling solutions. OLB is a method by which jobs are scheduled based upon the current loads on the machines. If there is an open or unloaded machine, OLB schedules a job to run on that machine. Put simply, it is a form of "queue management," whereby the queues are evenly loaded with no attention being paid to jobs already enqueued or the expected run-time of the same job on different machines (ETC). Another scheduling technique, which uses the ETC concept that was pioneered by SmartNet, is Limited Best Assignment (LBA). LBA considers one of the important parameters of scheduling, the expected performance of each job on the various architectures in the heterogeneous computing environment. LBA assigns each job to the machine upon which it is expected to execute the fastest [Ref. 1], assuming (unrealistically) that no other job is using that machine. Both OLB and LBA consider only half of the information that is required for the creation of a near-optimal schedule.

SmartNet considers both job performance and machine loads in its schedule creation. Armed with these two parameters, it develops a better schedule. Section F of this chapter provides examples of how a better schedule is generated using this information.

C.   SMARTNET'S PURPOSE
1.   Goal of SmartNet

SmartNet is a scheduling framework for distributed, heterogeneous, high performance computing (HPC). In this role, SmartNet strives to:

- Maximize computing power,

- Increase the throughput of a set of jobs,

- Optimize cost-effectiveness,

- Leverage existing resources, and

- Ensure robust scheduling.

In this context, the term "framework" means that SmartNet provides a mechanism that can enhance the performance of existing systems, such as DCEs or RMSs. As a framework, SmartNet was also designed so that it can easily accommodate new scheduling criteria and heuristics. This makes SmartNet a viable tool for a majority of HPC sites, regardless of the type of task and resource management that is currently utilized at that site. SmartNet can be applied to nearly any environment where the dynamics of the scheduling problem require a near optimal solution.

2.   Functionality

SmartNet is designed to allow a single administrator to manage the entire system. Users submit tasks to SmartNet. As tasks are received by the SmartNet server, they are placed into a database, a schedule is created or updated, and the tasks are run when the schedule indicates they should be. The database is a simple plain text file with a particular (and strict) format that is cached in memory when SmartNet is running. It is from this database that the server gets its job/machine estimated run-time (ETC values) information and to which the server adds new experiential information. This database information is the source of information for the construction of the schedule. Given the job/machine ETC values in the database, the scheduling algorithms are applied to create a near-optimal schedule. The server initiates the schedule and tracks the behavior of all jobs throughout the entire run-time process. If a job runs longer than anticipated, it can be terminated or flagged. Such a "rogue job" might cause an e-mail message to be generated from SmartNet to the original tasking entity, letting that group or user know that something was wrong with their job. As jobs complete, experiential data is collected and saved into a database. As experiential data is gathered, "learning" occurs, and SmartNet changes compute characteristic and expected time to complete (ETC) data in the database [Ref. 2].

14

D. SMARTNET ARCHITECTURE
1. SmartNet Processes

SmartNet is made up of several different processes, each with its own mission, yet relying upon messages to pass data between its processes. These processes include the Scheduler, the SmartNet Database, the Learning and Accounting Process, and the Controller. Messages exchanged consist of Requests, Control Information, and Data. Figure 4 depicts the relationships of these pieces.
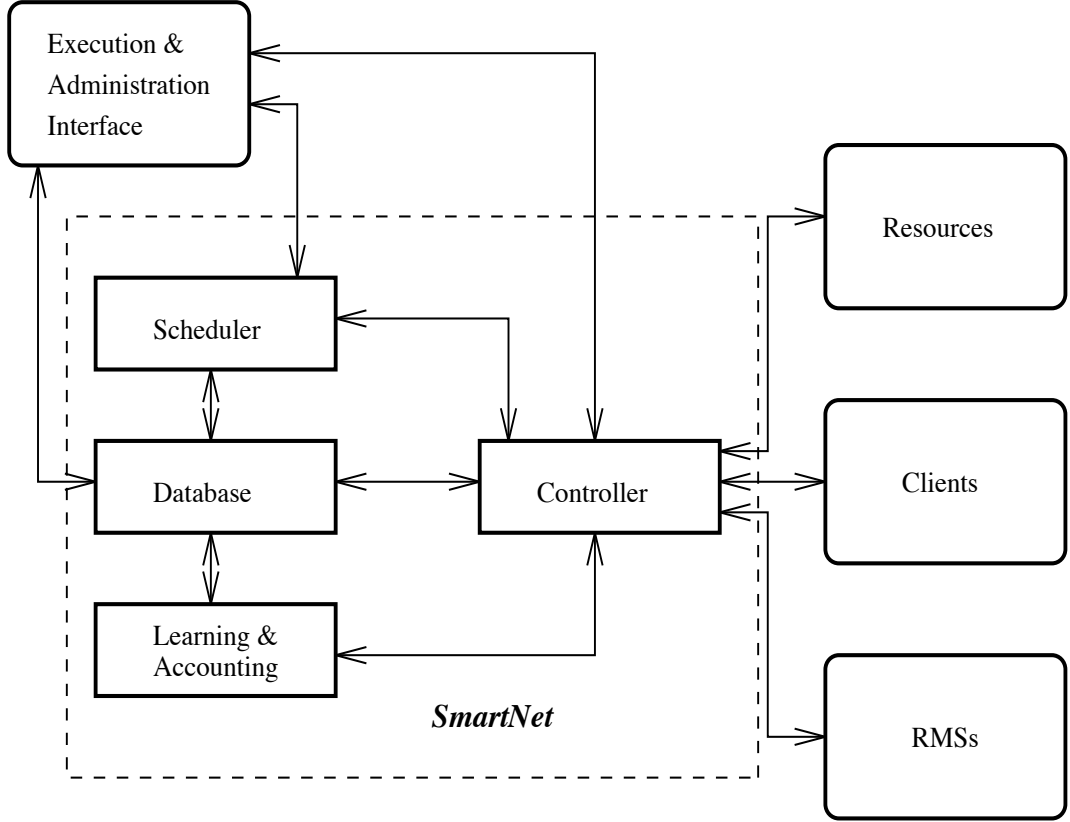
Figure 4. SmartNet Architecture, from [Ref. 2].

a. Interfaces

There are two user interfaces, one for the user who is submitting a job to be run and one for the SmartNet system administrator who oversees the proper operation of SmartNet. Graphical and command line versions exist for each. Users

15

can set priorities for their jobs, but the system administrator has ultimate control [Ref. 1].

b.   *The Controller*

The actual execution of jobs on resources may be controlled by any one of several facilities, including Resource Management Systems (RMSs), other versions of SmartNet, or Distributed Computing Environments (DCEs) [Ref. 2].

c.   *The Scheduler*

The SmartNet Scheduler contains both optimization and scheduling algorithms. There is a need for multiple algorithms because no polynomial algorithm optimally schedules for all environments. New schedulers can be added by the Smart-Net system administrator to take advantage of changing or unanticipated environments. Optimization is key to the performance of SmartNet. SmartNet can implement any number of optimization criteria, although only heuristics for maximizing the throughput by minimizing the completion time of the last job that finishes are present. Optimization criteria are what direct SmartNet to utilize specific search and scheduling algorithms. The algorithms built into the SmartNet scheduler are discussed in Section 2 [Ref. 2].

d.   *The Database*

The SmartNet database is an ASCII text file containing information about sites, groups, machines, models (jobs), and model-machine pairs. The database can be built or edited by hand, but the SmartNet Editor is a good tool to use, as it forces the administrator to input required data and writes the database in the proper format. SmartNet is not forgiving of improper formatting. As the database is parsed, data is evaluated and placed into objects commensurate with the order of data in the file [Ref. 10]. Appendix A shows the fields of the database and the information contained therein. Of particular importance is the expected time for completion (ETC) field in the model-machine listings. This ETC data is what SmartNet uses to create a schedule. The finish times of jobs must be either estimated by the programmer or

collected by SmartNet over the course of several runs in order for SmartNet to create anything close to a near-optimal schedule. Chapter IV contains detailed information about the changes that we made to this database, and to routines that read and write to the database, in order to perform our experiments.

e.   *The Learning and Accounting Process*

Presently, SmartNet's algorithms for learning and accounting are rudimentary. The framework exists, though, to permit easy integration of additional algorithms. As we mentioned in Section 2, rogue processes are tracked and reported. The action taken upon discovering a rogue process is specified by the user or system administrator at startup. Another form of learning and accounting that occurs is the gathering of experiential data after job completion. SmartNet will collect run-time statistics and write them out to the database file, making use of the information later during the scheduling and execution of similar jobs. [Ref. 1]

f.   *The Controller*

The Controller enters the picture when jobs terminate, jobs become rogue processes, new job requests are input, and when machines or networks go down. All of the above events may cause SmartNet to create a new schedule or re-start certain uncompleted jobs. The controller is designed to allow SmartNet to:

- allow redundancy in critical environments,

- operate in environments where resource availability is not guaranteed,

- be integrated with an RMS and provide scheduling assistance to that RMS, and

- coordinate the efforts of multiple RMSs [Ref. 1].

2.   SmartNet Algorithms

SmartNet uses a number of algorithms to create a schedule. The general characteristics of these algorithms are discussed below.

a. *Exhaustive Algorithm*

An Exhaustive Algorithm provides a "brute force" solution to the scheduling problem. Every possible data combination is generated and compared. Because this scheduling problem is NP-complete, this algorithm, that produces an optimal result, can only be used with very small data sets [Ref. 6].

b. *Greedy Algorithms*

Greedy Algorithms make the best local choice available at a specific point in the search tree [Ref. 6, pages 329–336]. For instance, if a Greedy algorithm is to choose the cheapest candy, and is searching a row of candy including a 75 cent Milky Way, a 55 cent Almond Joy, and a 35 cent package of Trident, it will choose the Trident over the other two. This appears to be an optimal solution; however, it is an optimal *choice*, based upon the candy considered at that point in the search tree. It is a best local choice. If a twenty cent box of Tic-Tacs lies on another row, it is the cheapest candy, and so the true optimal choice. Whether or not this decision aids in the production of an optimal solution depends upon the parameters of the entire problem. Since the Greedy Algorithms look for the best choice at some point in the search tree, complete consideration of the effects of the choice upon the end result are not made. Greedy algorithms are deterministic and produce only near-optimal results. SmartNet uses both an $O(mn)$ algorithm, which we call Fast Greedy, and an $O(mn^2)$ Greedy algorithm.

c. *Evolutionary*

Hartmut Pohlheim presents a fine explanation of evolutionary algorithms, portions of which are included here.

Evolutionary algorithms are stochastic search methods that mimic the metaphor of natural biological evolution. Evolutionary algorithms operate on a population of potential solutions applying the principle of survival of the fittest to produce better and better approximations to a solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads

to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation.

[I]t can be seen that evolutionary algorithms differ substantially from more traditional search and optimization methods. The most significant differences are:

- Evolutionary algorithms search a population of points in parallel, not a single point.

- Evolutionary algorithms do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the directions of search.

- Evolutionary algorithms use probabilistic transition rules, not deterministic ones.

- Evolutionary algorithms are generally more straightforward to apply.

- Evolutionary algorithms can provide a number of potential solutions to a given problem. The final choice is left to the user. (Thus, in cases where the particular problem does not have one individual solution, for example a family of pareto-optimal solutions, as in the case of multi-objective optimization and scheduling problems, then the evolutionary algorithm is potentially useful for identifying these alternative solutions simultaneously.) [Ref. 11]

d.   *Simulated Annealing*

Simulated annealing is a stochastic optimization method useful for finding global minimum cost configurations of NP-complete combinatorial problems with cost functions having many local minima [Ref. 12].

Simulated annealing builds on an analogy between the way metals contract with decreasing temperature into a minimum energy crystalline structure and the way searches for a minimum can be performed. After metal is heated and manipulated, it must be cooled. The best way to cool metals is to do it slowly. This allows the molecular makeup of the metal to slowly contract and "settle" upon itself which reduces the probability of cracks, "bubbles", and otherwise weak bonds throughout the entire mass of the metal structure. If metal is heated and then cooled very quickly, the contraction of the molecular structure tends to settle into local minima rather than to contract into a more stable, true minima. The metallurgic process of annealing then

19

compares to stochastic optimization methods like this: The heated metal is the random state that needs to be reduced to some sort of minima. In SmartNet, this would be the minimum time for completion of all jobs being scheduled. The temperature is a parameter that governs the probability of increasing the cost function at any step in the search for the global minima [Ref. 12].

The simulated annealing algorithm requires a valid solution space, a way to randomly move about in the solution space, a method for evaluating cost functions, and an annealing schedule. The annealing schedule includes the initial "temperature" variant and rules for decreasing that temperature throughout the search process. [Ref. 12]

Simulated annealing has several advantages. Specifically, simulated annealing:

- can deal with arbitrary systems and cost functions,
- statistically guarantees finding a near-optimal solution,
- is relatively easy to code, even for complex problems, and
- generally produces "good" solutions.

This makes simulated annealing an attractive, but computationally expensive, option for optimization problems where heuristic (specialized or problem specific) methods are not available. [Ref. 12]

e.   *Future Efforts*

As SmartNet is still a work in progress, there are continual efforts to develop better performing algorithms.

E.   SMARTNET PERFORMANCE

Previous work with SmartNet, detailed in [Ref. 1], provides the following information concerning schedules generated by SmartNet.

The performance data shown in Tables I and II was developed from several scheduling problems run on SmartNet in simulation mode. The scheduling problems

varied in both the number of jobs being scheduled and the number of machines available, as well as the amount of heterogeneity. The number of jobs and machines varied for each problem, but was always somewhere between two and 1000 jobs and two and 500 machines. The two modes of heterogeneity used were:

- Consistent Architectures. Given a set of machines, if one job runs faster on a particular machine, then <u>all</u> jobs will run faster on that particular machine.

- Mixed Architectures. Given a set of machines, one job running faster on a particular machine has no bearing on how other jobs might run on that particular machine. No generalizations about the performance of all the jobs on these machines can be deduced.

The algorithms were judged on how well they minimized the last job's completion time. Knowing that finding an optimal schedule is an NP-complete problem [Ref. 1], the baseline used for comparison was derived from a lower-bound algorithm. This algorithm does not produce a valid schedule, but does obtain a time known to be less than the time at which the last job will complete.

Table I provides average time of completion of the last job in a schedule for a variety of architectures and algorithms. The numbers represent time, and show that the schedule produced with a SmartNet Greedy algorithm (MinMin) is better than either the OLB or LBA generated schedules.

	SCALABLE ARCH.	ARCH. MIX	ARCH. MIX
JOBS/MACHINES	500/100	500/100	1000/500
LBA	100	86.2	422.6
OLB	5.47	4.01	7.33
MINMIN (SMARTNET)	3.78	3.14	4.01

Table I. SmartNet Performance: Average values for the time t at which the last job in a schedule completes.

Table II shows OLB, LBA, and SmartNet's Greedy algorithms' performance relative to a lower bound. After normalizing to the lower bound, the table shows

that given a 500/100 job/machine ratio on mixed architectures, the SmartNet Greedy algorithms completes six percent slower than the best possible time. OLB completes 28% slower than this time. LBA, on the other hand, completes 2,650% slower than this time.

	SCALABLE ARCH.	ARCH. MIX	ARCH. MIX
JOBS/MACHINES	500/100	500/100	1000/500
LBA	26.5	27.5	105.5
OLB	1.45	1.28	1.83
MINMIN (SMARTNET)	1.13	1.06	1.29

Table II. SmartNet Performance: Average values of t compared to our lower bound. t is the time at which the last job in a schedule completes. Our lower bound is represented as 1.00. This table shows that when SmartNet schedules 500 jobs on 100 mixed-architecture machines, the schedule is completed in six percent more time than our lower bound. From [Ref. 1].

F.   EXAMPLES

These examples help explain both how SmartNet works and how a knowledge of both machine load and anticipated job performance can create a better schedule.

We consider the following scenario: There are three machines, `Machine-A`, `Machine-B`, and `Machine-C`. Each machine is of a different architectural design (SIMD, MIMD, and Vector, respectively). There are four jobs, `Job1`, `Job2`, `Job3`, and `Job4`, each with different compute characteristics. Table III provides ETC values for the job-machine pairs.

1.   Example 1: Opportunistic Load Balancing

OLB is a method by which jobs are scheduled based upon the current loads on the machines. Figure 5 shows one possibility of how an OLB scheduler might schedule jobs to run on several machines. In this scenario, the OLB algorithm places the next job in the queue of the next available machine. If the jobs are ordered in the queue according to increasing job ID order, and if machines become available in the order

22

	Machine-A SIMD	Machine-B MIMD	Machine-C Vector
Job1	33	5	22
Job2	2	49	56
Job3	13	12	17
Job4	15	3	9

Table III. Job Run-times used in all examples.

`Machine-A`, `Machine-C`, `Machine-B`, and `Machine-B`, the jobs will be scheduled as in Figure 5. We note that the time of completion for all jobs is **56**.
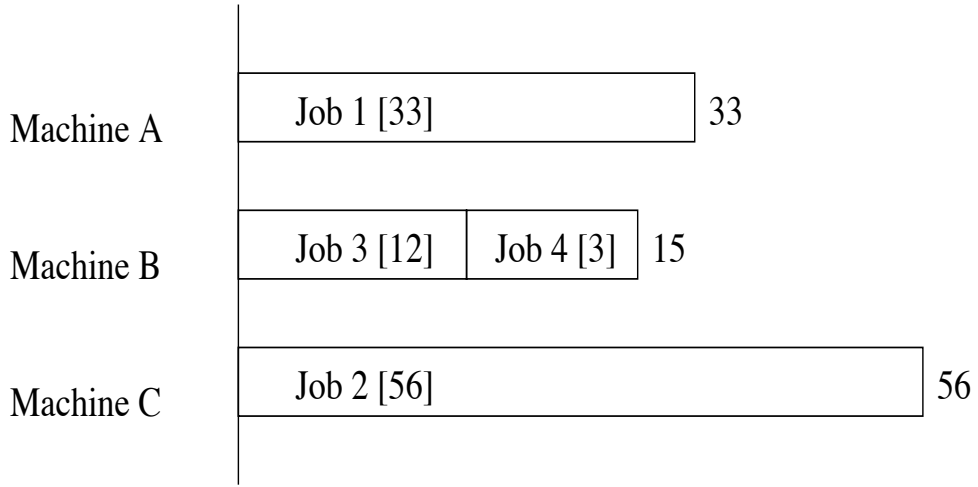
Figure 5. Example 1: An OLB Schedule.

2.   Example 2: Limited Best Assignment

LBA schedulers assign jobs to machines based upon the expected job's performance on each of the machines. In other words, the jobs are assigned to the machines upon which they should perform the best (i.e., have the shortest expected run-time) [Ref. 1]. We note that this algorithm assumes that each job that it schedules is the only job in the system. Again, Table III provides the expected run-time data used in this example.
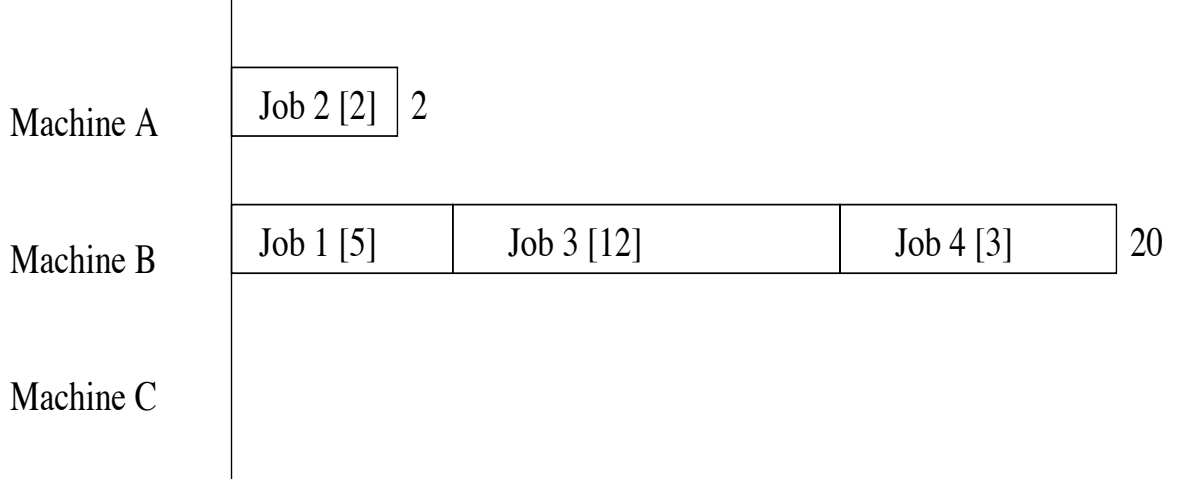
Figure 6. Example 2: An LBA Schedule.

Figure 6 shows how an LBA scheduler would schedule the four jobs on the three machines. We note here that the expected time of completion for all jobs is **20**.

3. Example 3: Greedy Algorithm

This example uses a Greedy Algorithm. This algorithm takes into account both machine loads (like OLB) and run-time performance (like LBA) to produce a near optimal schedule. Again, Table III provides the expected run-time data used in this example.

Figure 7 provides a SmartNet schedule for the Table III data. Here, the earliest expected run-time completion for all jobs is **15**. This is significantly better than either the OLB or LBA schedulers from Examples 1 and 2.
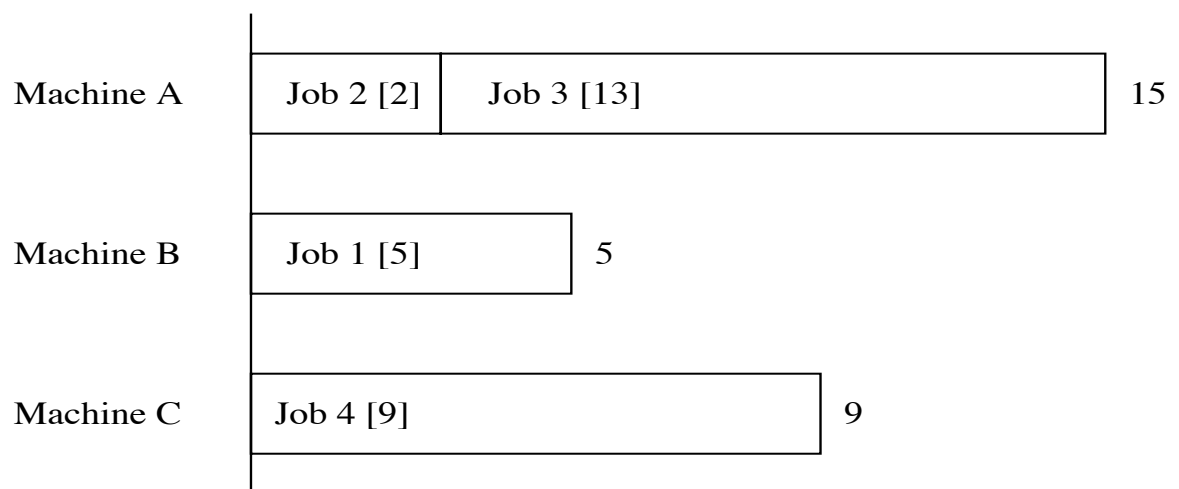
Figure 7. Example 3: A SmartNet Schedule.

III.   DISCRETE EVENT SIMULATION

A.   INTRODUCTION

This chapter explains discrete event simulation. Section B provides background information concerning simulation in general and explains why discrete event simulation is a useful tool. Section C describes discrete event simulation in detail. Random variates are explained in Section D. Section E presents concluding remarks.

B.   BACKGROUND INFORMATION

The desire to predict the performance of a system has led to the need to study both the system's performance and behavior. This desire is the driving force behind much academic and industrial research. In this context, a system might be:

- an actual mechanical entity, such as an automobile or a building,

- some measurable non-mechanical entity, such as a hurricane or an ecosystem, or

- a process or sequence of events involving both human and mechanical functions similar to the logistic example posed in Chapter I.

One characteristic common to the types of systems listed above is that they possess measurable parameters that influence their behaviors. For example, an automobile has the variable parameters velocity and acceleration, as well as the constant parameters weight, mass, and coefficient of friction. Performance of an automobile is affected by all of the above parameters. Parameters may be restricted to a designated range. A study of an automobile's performance would utilize these variable and constant parameters, as well as any restrictions in effect, and provide performance predictions specific to the input parameters. Such a study would be helpful in determining how an automobile might perform, given modifications to its weight or coefficient of friction. There are several methods available to study this or any system. While, in this case, the most obvious would be to study an actual automobile, there

27

are severe limitations to this method. It would be difficult, if not impossible, to make adjustments to the coefficient of friction without altering the shape of the automobile. Changing the shape of an automobile is difficult. The need to change the coefficient of friction, for example, limits the utility of experimenting on the automobile itself. In this case, and for many other types of systems, it is probably easier to construct a model. Figure 8 shows the different ways systems can be studied.
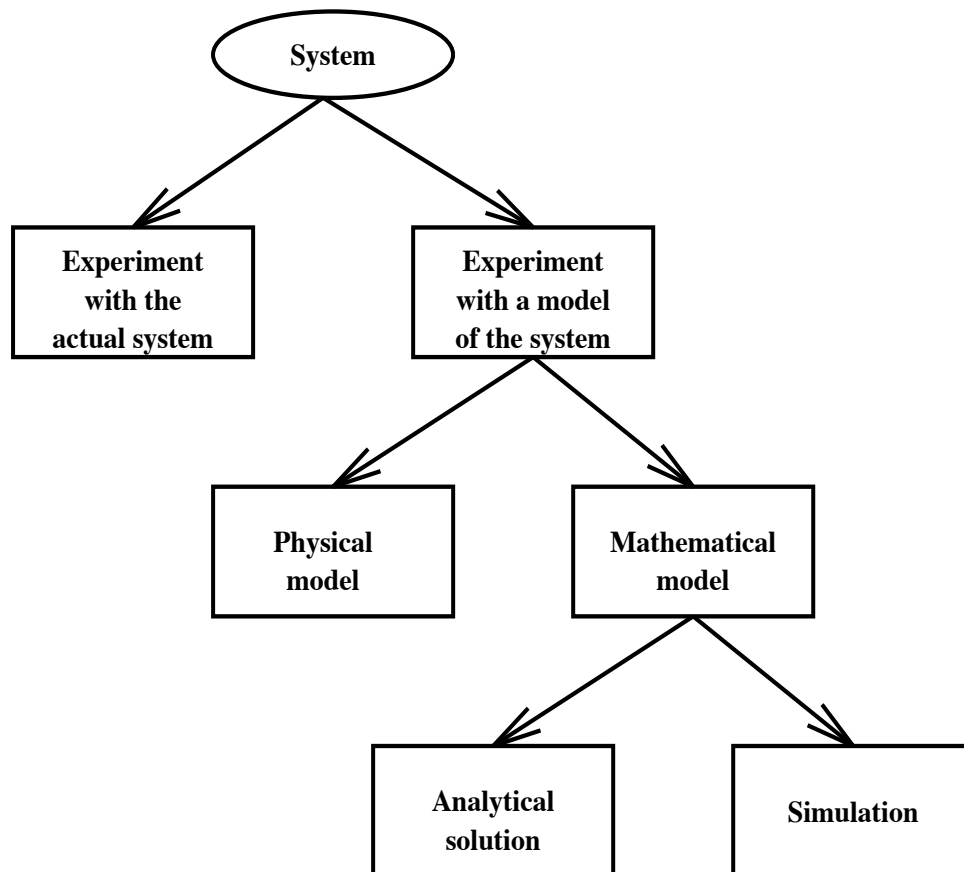
Figure 8. Ways to study a system, from [Ref. 13, page 4].

A model of a system can be constructed either mathematically or physically. Depending upon the complexity of the system, both can be difficult. There are obvious limitations and difficulties associated with constructing a physical model of a logistic system used to move troops and equipment from the United States to a foreign area

of operations[1]. Physical modeling would involve scaling a global problem down to a manageable size. In a high fidelity physical model, every physical feature of the logistic operation might be physically rendered. Physical features requiring duplication in this case would include the loading of ships and aircraft, troop movements, and airfield operations. The difficulty in making such a model accurate is obvious. Physical modeling to a reduced scale also introduces inaccuracy in many areas, not the least of which is the non-linearity of design characteristics between full and reduced size entities.

An alternate approach to physical modeling is mathematical modeling. Any physical system can be reduced to a mathematical model that represents those aspects of the system that the modeler desires to measure and control. In our logistic example, the loading of aircraft can be mathematically modeled as taking a deterministic amount of time dependent only on the type of cargo being loaded. Transit time can be modeled also as a deterministic amount of time, perhaps by using the average of historical data. Actual cargo can be modeled using its weight, mass, and measurement parameters and considered a "puzzle piece" to be moved, shifted, and transported in accordance with the priorities provided by the force commander. In general, a mathematical model is an order of magnitude less expensive model to produce than the physical model. Additionally, the designer can easily modify the fidelity of the various aspects of the system that are deemed important.

There are two methods for studying mathematical models: analysis and simulation. The analytical approach to studying a model requires the solution of mathematical equations. If the system being modeled is complex, though, it may be impossible to develop mathematical equations that consider the combined effects of every interrelated or critical piece of the model. Increasing the accuracy, or fidelity, of the model may require very complex mathematical equations. As an example, we consider modeling, in great detail, the logistic example from Chapter I.

---

[1]See example provided in Chapter I.

To model a single fork lift, we would need to mathematically represent such things as the mean time between failure of the engine, the fork lift mechanism, and the tires. Also, rate of failure of the operator, driving speed, lifting speed, haul rates, machine-to-task suitability, fuel consumption rate, and maintenance schedules would need to be modeled. As we see, there are numerous details in modeling a single fork lift, and the fork lift itself is only a single, small part, of the entire logistic system. There may be four different types of fork lifts at a single airport, and a total of forty fork lifts, altogether. The complexity of modeling forty fork lifts is greater than one fork lift, but even if modeling them is easy, forty fork lifts as a whole are *still* a small but vital piece of the logistic system. Further, more complex pieces of the logistic system would need to be included in the model.

- **Fuel.** There are some finite number of refueling trucks, as well as a finite amount of jet fuel. The delivery of fuel to the airfield, the process of refueling aircraft, and the performance characteristics of the personnel and machines involved in the entire refueling process would need to be modeled.

- **Scheduling.** Scheduling is an NP-complete problem. The airfield has a maximum physical capacity. The airfield also has a maximum workload under which it can operate. Every asset at the airfield needs to be scheduled so that the process of getting personnel and equipment onto aircraft and subsequently overseas works in accordance with the intent of the commander. Introducing scheduling into an analytical model may make it too complex to find a closed form solution.

- **The Human Factor.** In every environment where people are working under stressful conditions, accidents occur. When medium and large scale machinery are present, severe accidents are possible. Accident and injury rates must be modeled. Further, the consequences of these same accidents and injuries must also be modeled. For example, we consider the effect that the following scenario might have on the operation of an airfield: A Heavy fork lift operator is loading an extremely large metal storage container on a C-5 cargo plane. The C-5 is also being refueled. The fork lift operator has a heart attack and loses control of the fork lift. The fork lift drives the storage container through the side of the C-5, wrecking the jet's extensive hydraulic system. The refueler operator, seeing the situation, performs an emergency disengage of the refueler from the aircraft. His refueler dumps 500 pounds of highly flammable jet fuel on the tarmac. We see that such scenarios, when modeled with great fidelity,

are mathematically very complex. The individual effects may be easy to model; however, the comprehensive effect of the individual events may not be easy to model.

If, when using the analytical approach, very realistic assumptions and high fidelity are required, closed form solutions may be impossible, forcing the mathematical modeler to make simplifying assumptions that can cause the results to be useless. Suppose that the probability of a devastating accident involving a C-5 aircraft on an the ground during refueling is 0.0001. Further, it is known that the probability distribution is Gamma(0.0001, 15). If an accident of this type occurs, the airfields cycle rate of aircraft is decreased by 10%. The Gamma distribution does not have a closed form with these parameters. The mathematical modeler might choose to represent the probability of this event occurring, then, with an exponential distribution, because it has similar characteristics to the gamma distribution, and the exponential distribution and its inverse are both closed form expressions. Because of the need to simplify the mathematical model, the model no longer provides the desired accuracy, which may result in incorrect performance estimates.

An important part of modeling is simplification. Simplification is a method of reducing or removing specific complex factors which can be accounted for by other means. Using the fork lift example above, if, in reality, the fork lift breaks once every 10,000 hours, the modeler may be able to assume that the fork lift will not break. Ample consideration must be given to the possibility of skewing the results obtained from the model because of poor simplifying assumptions. If the fork lift actually breaks once every 10 hours, that factor would probably need to be included due to the frequency of occurrence.

A simulation, executed on a computer, also uses a mathematical model. When building simulations, it is easy to increase the fidelity of certain aspects of the system while decreasing the fidelity of others. We again consider the fork lift discussed above. A simulation model of a fork lift may not need to model fine details such as the mean time between failures, fuel consumption, lifting speed, and maintenance schedules. It

may make sense to consider all of these factors as one and model the work performed per hour. Such a simplification would reduce the complexity of the model, and might make it easier to evaluate. Simulation models are evaluated via their state variables. State variables are those parameters that are required to describe the model (and so, the system) at a particular point in time.

Simulation models can be classified along three dimensions:

- Static versus Dynamic. A static model is a snapshot of a system at a particular time, while a dynamic model is evolutionary.

- Deterministic versus Stochastic. A deterministic model has no random components. Output is a deterministic function of input. A stochastic model is, in contrast, non-deterministic.

- Continuous versus Discrete Time. A continuous time model is one in which the state variables change continuously over time. A discrete time model is one for which the state variables change instantaneously at separate (discrete) points in time.

- Continuous versus Discrete States. A continuous state model is one in which the values of the state variables can take on any of a defined range of values. A discrete state model is one in which the values of the state variables are restricted to a subset of acceptable values.

The type of simulation used to provide results in this thesis is static, stochastic, and discrete in nature. This type of simulation is commonly called discrete event simulation.

C.   DISCRETE EVENT SIMULATION
1.   Overview

Discrete event simulation models a system's activity as it progresses through time. The operation of a system can be thought of as a collection of events that make up the system's activity. An event is "any instantaneous occurrence that may change the state of the system." [Ref. 13, page 7] Events occur at different times, and are stamped with the time at which they occur. The state of the system is, informally,

its current condition. System state is defined by system specific state variables that describe the system's condition [Ref. 13, page 81]. As events that are to occur in the future are generated as a byproduct of simulating a current event, they are stored in an event queue, where they stay until the simulation clock advances to the time of their occurrence. Events in event queues are often ordered according to the simulation time at which they are to occur. As the discrete event simulation progresses, individual events are taken out of the event queue and processed. When an event is removed from the event queue for processing, the simulation clock is advanced to the time stamp on that event.

Discrete event simulation characteristically requires three sets of variables.

- Time variable t. t is used to track elapsed simulation time and is also called the simulation clock.

- Counter variables. These are used to track repetitions of certain events and the time that they occur.

- System state variables. These are model/system dependent; they describe the state of the system at any given time [Ref. 14, page 81].

The advancement of time in discrete event simulation can be a difficult concept to understand. The elapsed simulation time and the actual time required to run a simulation are usually different. The time required to run a simulation may be greater or less than the elapsed simulation time, and is dependent upon the particular model. An example of a model where simulation time would probably be greater than real time is in the simulation of subatomic particle movement. An example of a simulation that would probably require less time than real time is simulation of continental drift. Advancement of the simulation clock is usually done via one of two methods:

- Next-Event time advance. Time is advanced whenever an event occurs.

- Fixed-Increment time advance. Time is advanced at fixed intervals.

Next-Event time advance is the most prevalent method [Ref. 13, pages 7—9].

Figure 9 depicts the flow of control for a next-time advance discrete model.

33
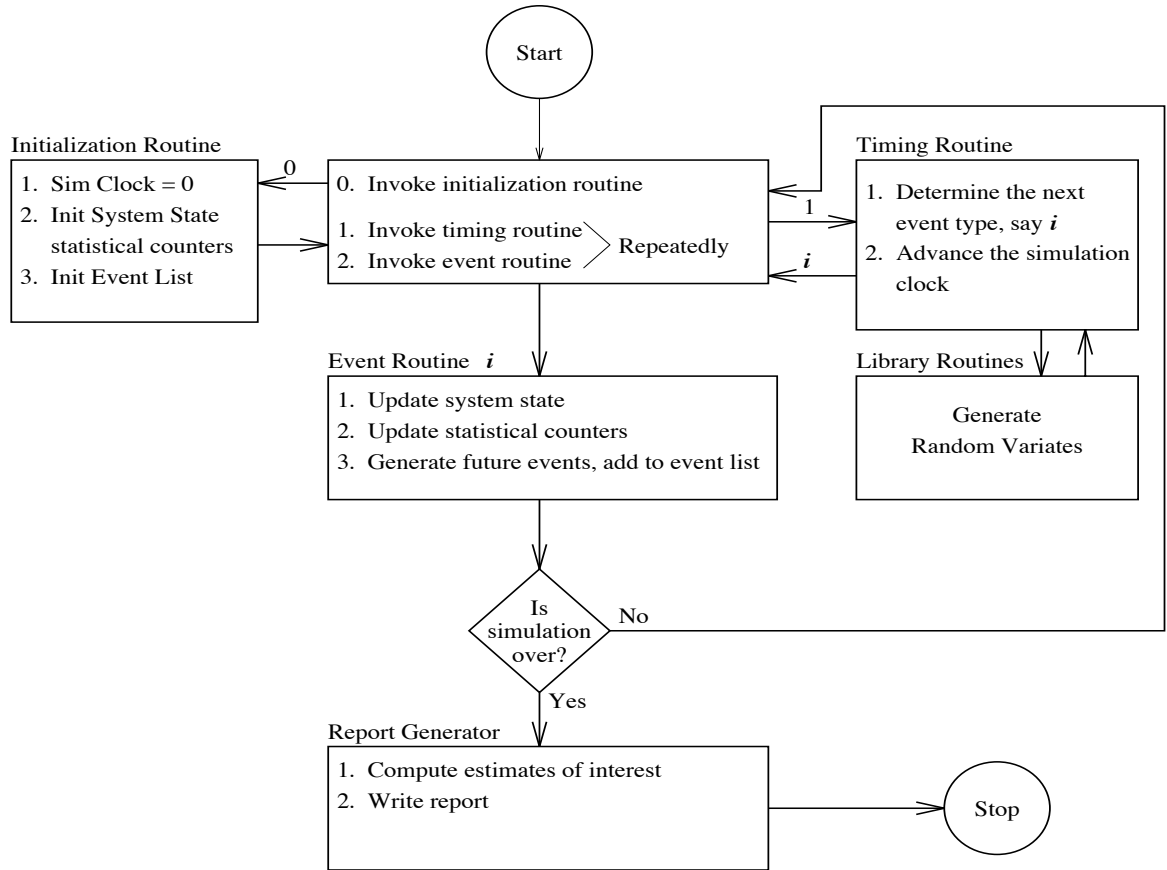
Figure 9. Flow of control in Discrete Event Simulation, from [Ref. 13, page 12].

2.   An Example of Discrete Event Simulation

Discrete event simulation can be applied to the logistic system described in Chapter I. The mission to be accomplished, using the logistic system, is the efficient movement of troops and supplies from various locations throughout the United States and other allied nations to some foreign area of operation. This system provides numerous examples of the difficulties found when building a near optimal schedule for the use of logistic assets. It is also a good system to demonstrate the utility and suitability of discrete event simulation. Of particular note, however, is the difficulty of modeling any system this complex and large. Akin to this difficulty is the need for specific problem statements. In other words, we need to know what we are modeling and why. It is often infeasible to model every aspect of such a system with great

34

fidelity, as the size of the system, including dependencies between subsystems, would be too complex.

Eastern Africa Area of Operation

Air Transport Flow Into and Out Of
The Area of Operation
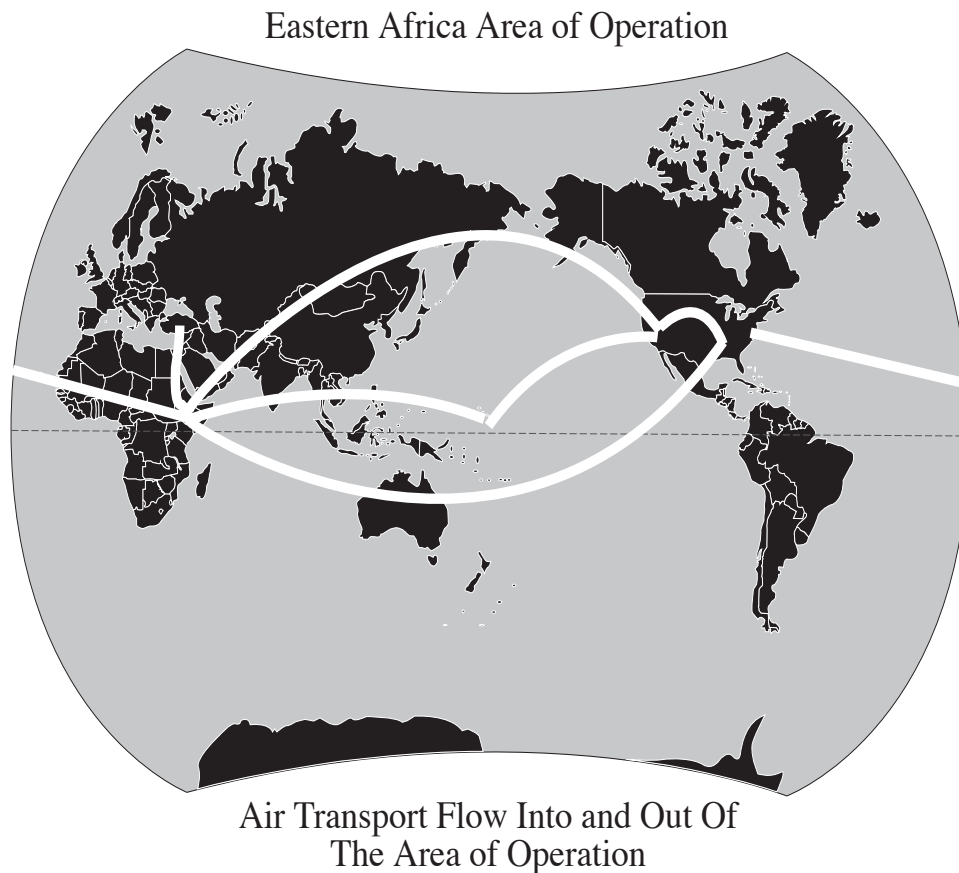
Figure 10. Logistic Example: Air transport assets into and out of Somalia.

An important factor in the success of a logistic system is the capability, performance, and scheduling of air transport assets. Whenever U.S. forces deploy to foreign soil for both peace keeping and combat missions, multiple plans for troop and equipment build-up in that area are developed. The plans include rosters of units (troops and equipment) that will be deployed and schedules designating when the units are to arrive. The deployment of forces can take from several days to several months in order to reach the force structure needed to fulfill the requirements of the mission. The theater commander will be very concerned about reaching his desired in-theater force structure, as it will drive his ability to begin, continue, and complete

35

the mission. The logisticians must plan the movement of assets into the area of opera-
tions as efficiently and effectively as possible to allow the theater commander to mass
his forces appropriately. Transportation of equipment and troops by air can help meet
initial force build-up requirements both efficiently and quickly.

The commander's desires specific to air transport scheduling and availability
can be simulated using discrete event simulation. Two important questions that the
simulation must answer for the commander are **"How long will it take for my
forces and their equipment to be transported into the area of operations?"**
and **"Given the planned scenarios, which one most rapidly places the major-
ity of my fighting forces and their equipment on the ground?"** One approach
to answering the commander's questions is to build a computer model and simulate
the movement of each force structure into the area of operation, and report the length
of time required. The goal is to use the simulation as one of the many tools available
to the commander.

Discrete event simulation has direct application to modeling the flow of aircraft
into an area of operations. We consider the following pseudo-algorithm:

- **loop begins**

  1. $Aircraft[aa]$ arrives at $fromUSAirfield[bb]$
  2. $Aircraft[aa]$ is ready to be unloaded
  3. $Aircraft[aa]$ is ready to be loaded
  4. $Aircraft[aa]$ is loaded
  5. $Aircraft[aa]$ departs $airfield[bb]$ for $AREA\_OF\_OPERATIONS$
  6. $Aircraft[aa]$ arrives at $AREA\_OF\_OPERATIONS$
  7. $Aircraft[aa]$ is ready to be unloaded
  8. $Aircraft[aa]$ is ready to be loaded
  9. $Aircraft[aa]$ is loaded
  10. $Aircraft[aa]$ departs $AREA\_OF\_OPERATIONS$ for $toUSAirfield[cc]$

- **loop ends**

The above list enumerates several events that a discrete simulation of the system might incorporate. The dynamics of this problem dictate that the above ten events must occur at some point, and in the stated order, during every round trip flight of an aircraft ($Aircraft[aa]$) from the United States ($fromUSAirfield[bb]$) to a foreign airfield ($AREA\_OF\_OPERATIONS$) and back to the United States ($toUSAirfield[cc]$).

The "discrete event" aspect of the simulation refers to the time interval between specific events. The amount of time advanced is dependent upon what is going on in between the two events. While a detailed discussion of a discrete event simulation for the above example is beyond the intent of this section, an explanation of what occurs between two of the events will suffice. We consider the events in lines 1 and 2 above:

1. $Aircraft[aa]$ arrives at $fromUSAirfield[bb]$
2. $Aircraft[aa]$ is ready to be unloaded

Event 1 is labeled with the time ($Simulated\_time\_1$) that an aircraft arrives at a U.S. airfield. Event 2 is labeled with the time ($Simulated\_time\_2$) that the same aircraft is ready to be unloaded. The duration between event 1 and event 2, in reality, is determined by the amount of time the aircraft is idle on the ground, which is effected by the number of other aircraft already on the ground as well as the rate at which those aircraft can be unloaded. The duration between events 1 and 2 in the simulation is either deterministic or stochastic. $DeltaT$ represents the time required to unload the aircraft. The advanced time function might proceed as follows.

1. $SIMULATION\ CLOCK = Simulated\_time\_1$
2. $Simulated\_time\_2 = Simulated\_time\_1 + DeltaT$
3. $SIMULATION\ CLOCK = Simulated\_time\_2$

In our example, $DeltaT$ is determined by a distribution that is based upon observed data. If an aircraft must always wait the same amount of time before being unloaded

37

after it arrives at the airfield, a constant could be used for $DeltaT$. If the amount of time that an aircraft must wait to be unloaded after it lands at the airfield is not fixed, the probabilistic nature of that duration must be recreated for the simulation. Recreation of this random process requires the following:

- The identification of the mathematical distribution that matches the distribution of times that the aircraft must wait to be unloaded.

- The generation of a random variate[2], $DeltaT$, from the mathematical distribution previously identified.

The strength of discrete event simulation is evident when the simulation is actually performed. Actually loading and unloading the aircraft may require several days. However, because discrete event simulation instantaneously advances simulated time to the time of the next event, the simulation may only require several seconds. The $SIMULATION\ CLOCK$ is advanced at each event by the appropriate real world $DeltaT$, and the simulation terminates with realistic results in significantly less time than the actual sequence of events.

D.   RANDOM VARIATES

The very nature of discrete event simulation requires it to incorporate stochastic processes to account for the inherent randomness in the system. We again consider the logistic example used throughout this chapter. While the process of moving troops, supplies, and equipment from the United States to a foreign shore is a highly scheduled, well planned operation, there is unavoidable randomness in the system. As an example, we consider the effect of mechanical failure on air transport flow. Data, such as the time between failures, can be gathered for the relevant aircraft. This data can then be analyzed statistically to determine the mean and variance, and a distribution fitted to the failure rates. Using this information, the failure can be simulated so

---

[2]Random variates are explained in Section D.

as to occur randomly according to a distribution that has been fit to the observed data. The simulation, then, is capable of demonstrating the effect of a decrease in the movement rate of aircraft into the area of operations. Further simulation work may include modeling how the logistician or commander adapts to the lost air transport movement capability and implements an updated flow plan.

1.   Random Versus Pseudo-random Numbers

Knuth provides a good definition of the term *random*.

[The idea of randomness often invokes] philosophical discussions about what the word "random" means. In a sense, there is no such thing as a random number; for example, is 2 a random number? Rather, we speak of a *sequence of random numbers* with a specified *distribution*, and this means loosely that each number was obtained merely by chance, having nothing to do with other numbers of the sequence, and that each number has a specified probability of falling in any given range of numbers. [Ref. 15, page 2]

After computers were introduced, people began looking for efficient ways to obtain random numbers using computer programs. Several methods were investigated, but none proved efficient nor simple enough to gain acceptance. These problems led to an interest in the production of random numbers using the arithmetic operations of computers. John von Neumann suggested the "middle-square" method in 1946. The idea is to take a number chosen at random, square that number, then extract the middle digits to produce the next random number. The problem with this method is that there really is not any randomness in the process. Each number is completely determined by the one before it. However, the sequence of numbers *appears* to be random. The generation of sequences of random numbers deterministically is usually called *pseudo-random* number generation. Within most textbooks, as well as in this thesis, sequences are termed random, with the understanding that sequences only appear to be random. [Ref. 15, page 3]

If a random sequence of numbers is generated deterministically, that sequence can then be reproduced. Is this ability to reproduce a sequence of numbers from

a random number generator really undesirable, though? In many cases, it is, in fact, desirable. There are many occasions where the precise behavior of a simulated stochastic process might need to be reproduced multiple times. The only way to do this is to reproduce the sequence of random numbers used previously. This technique is particularly useful during debugging, when the performance of the simulator may need to be consistent in order to rule out anomalous factors. [Ref. 13, page 424]

2.   Random Variates and Distribution Characteristics

A random variate is a random observation generated from a probability distribution [Ref. 13, pages 11, 462]. A probability distribution has specific characteristics that are referred to as the first, second, and third moment. Table IV shows the parameters that characterize several well known types of distributions.

DISTRIBUTION	PARAMETER 1	PARAMETER 2	PARAMETER 3
GAUSSIAN	MEAN	VARIANCE	NA
EXPONENTIAL	MEAN	NA	NA
UNIFORM	SMALLEST LIMIT	LARGEST LIMIT	NA
WEIBULLGAMMA	SHAPE PARAMETER	SCALE PARAMETER	NA
LOGNORMAL	SCALE PARAMETER	SHAPE PARAMETER^2	NA

Table IV. Parameters of Various Distribution Functions.

We will use the Gaussian (Normal) distribution as an example in this section. Figure 11 shows a histogram of a Gaussian distribution of 100,000 random variates distributed around a mean of 100 with standard deviation 15. Random variates can be thought of as the x-axis values. The frequency of x-axis values is plotted along the y-axis. The Gaussian curve shows us that there are more random variates near the mean, and fewer as you move away from the mean. An explanation of how random variates can be generated from this information can be found in Section 3.
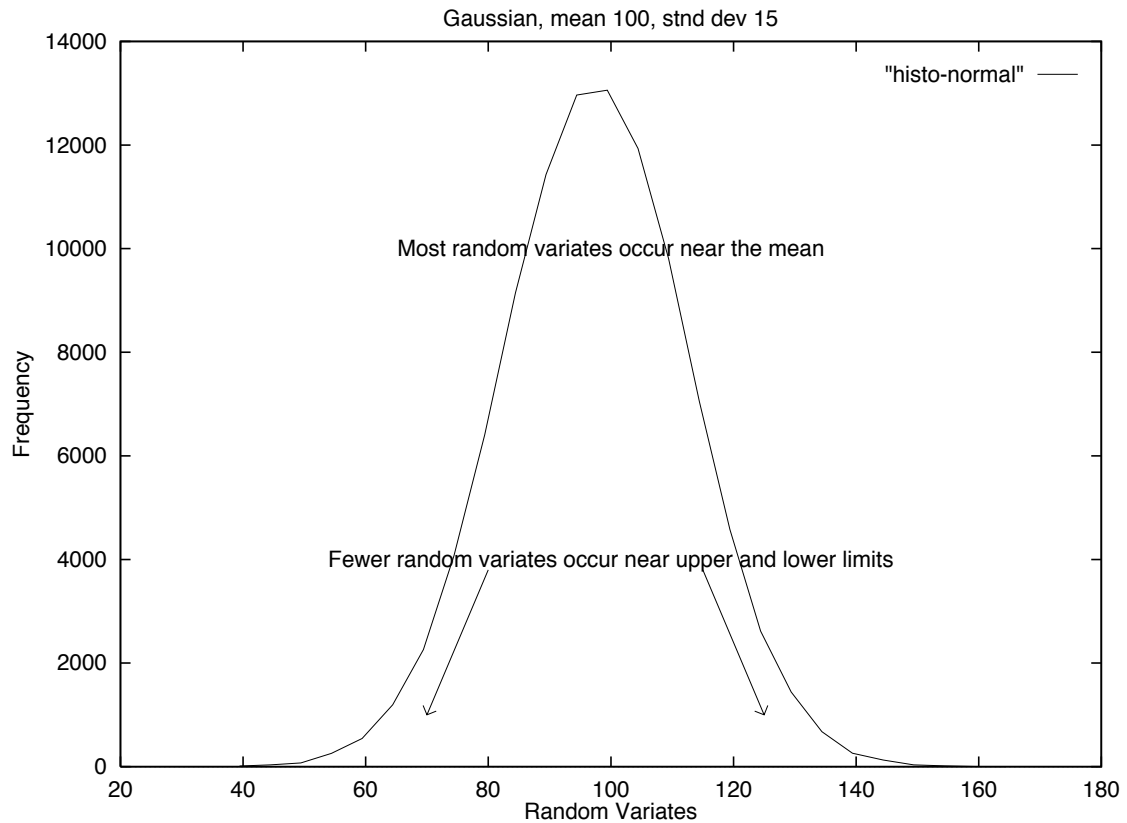
Figure 11. An Example of a Gaussian Distribution, mean of 100, standard deviation of 15.

3.   Generating Random Variates

First, we present a short summary of what we have discussed thus far. A stochastic process is a process that contains some probabilistic components. In order to accurately simulate a stochastic process, those aspects of the process that occur randomly must retain their random nature in the simulation. In order to simulate a stochastic process, then, specific information about the nature of the random factors must be known.

For example, we again consider the fork lift. We assume that the rate at which the wrong cargo (in error) is loaded on an aircraft is a random parameter that must be considered in a simulation of the fork lift. Experimental data may show that the mean time between a loading error per fork lift is 100 hours, where the data from

which this information was gathered behaves as a Gaussian (Normal) distribution with mean 100 and standard deviation 15. This example was used to produce Figure 11. Given this information, the simulation of the fork lift can incorporate a random error corresponding to this known behavior. Instead of a constant value of 100 hours for the mean time between a loading error, a factor can be added to the simulation that causes the fork lift to load the wrong cargo randomly, but at time differences generated from a Gaussian distribution of mean 100 and standard deviation 15.

The mechanics of generating random variates are specific to the distribution in question; however, every method relies upon a source of independent and identically distributed (IID) random variates uniformly distributed on the interval $(0, 1)$ [Ref. 13, pages 462-463]. These are commonly called IID U(0,1) random variates. The most important aspect of generating random variates, then, is a valid source of IID U(0,1) random variates. While there are numerous random number generators available for particular languages and operating systems, the user must ensure that the random number generator they choose to use is in fact IID U(0,1).

There are several general classes of approaches for generating random variates from an IID $U(0, 1)$ generator.

- **Inverse Transform.** This method is best used for generating random variates with a distribution function F that is continuous and increasing when $0 < F(x) < 1$. The technique is to generate $U \sim U(0,1)$ and return random variate $X = F^{-1}(U)$. [Ref. 13, pages 465-474]

- **Composition.** This technique applies when the distribution function can be best expressed as a combination of other distribution functions. When the distribution function F can be expressed as a convex combination of distribution functions $F_1, F_2, \ldots, F_n$, it may be easier to gather sample random variates from the $F_i's$ than from the original F. [Ref. 13, pages 474-475]

- **Convolution.** The term convolution "comes from the terminology in stochastic processes where the distribution of X is called the *m-fold convolution* of the distribution of Y_j." [Ref. 13, page 477] This technique is best suited for distributions for which the generation of random variable X is more easily expressed as a sum of several IID random variables. The implementation of this technique involves the generation of $Y_1, Y_2, \ldots, Y_k$, IID, each with distribution function

42

F, and the subsequent return of random variate $X = Y_1 + Y_2 + \ldots + Y_k$. [Ref. 13, page 477-478]

- **Acceptance-Rejection.** This is a less direct approach than the aforementioned techniques, yet is still useful, particularly when a more direct method is too difficult or costly. This method requires the specification of a function t that majorizes[3] the density function f. This technique involves generating a Y that has density r, and generating a $U \sim U(0,1)$, that is independent of Y. If $U \leq \frac{f(Y)}{t(Y)}$, this method must return the random variate $X = Y$, otherwise, it generates a new value and similarly tests it. [Ref. 13, page 478]

The method used to generate a random variate should be chosen based upon the particular distribution the random variate is to be drawn from, and the ease and reliability with which random variates can be generated for that distribution. The generation of random variates is considered reliable if the occurrence of individual random variates is statistically equivalent to the distribution from which they are derived.[Ref. 13, page 463]

If the distribution is of a known type, implementations are readily available that require little work and promise the accurate generation of random variates. Otherwise, the easiest method to implement is most likely Inverse Transform. Inverse Transform can be an easy method because random variates are generated from the inverse of the distribution function F; inverting the distribution function may be a simple task. However, for some distributions, the inverse may be undefined. For example, the Gaussian distribution function cannot be inverted because it does not have a closed form expression [Ref. 13, pages 465—466]. While there are numerical methods to evaluate F^{-1} when there is no closed form, such an Inverse Transform may not be the most computationally efficient method to use. If the distribution in question is multi-modal, or a combination of two or more different distributions, random variate generation becomes more difficult, and Composition or Convolution should be used.

---

[3]Majorizes: $t(x) \geq f(x)$.

a. *Generating Gaussian Random Variates*

The Gaussian distribution is characterized by the first moment (mean) and second moment (variance). A random variate $X \sim N(0,1)$ can be used to obtain some $X' \sim N(\mu, \sigma^2)$ by setting $X' = \mu + \sigma X$. The ability to generate this data from the first and second moments is helpful, because it allows us to focus on obtaining standard Gaussian random variates ($N(0,1)$). Random variates particular to *any* Gaussian distribution can be obtained using the above computation.[Ref. 13, pages 490-491]

There are two commonly used methods for obtaining standard $N(0,1)$ random variates. The first is the Box and Muller method, which is effective but has a limitation when used with linear congruential random number generators(LCGs). (LCGs are explained below.) We now explain the Box and Muller method, and then explain this limitation. The Box and Muller method begins by generating two random variates, U_1 and U_2, from an IID $U(0,1)$ generator. The variables X_1 and X_2 are generated using the following formulae.

$$X_1 = \sqrt{-2 \ln U_1} \cos 2\pi U_2$$
$$X_2 = \sqrt{-2 \ln U_1} \sin 2\pi U_2$$

X_1 and X_2 are then IID $N(0,1)$ random variates. The limitation alluded to above can be easily seen when U_1 and U_2 are not true IID $U(0,1)$ random variables, but are dependent, which might can occur if U_1 and U_2 are generated using the same seed. Linear congruential generators rely on recursion to generate numbers. The recursive formula for a linear congruential generator is as follows.

$$Z_i = (a Z_{i-1} + c)(mod\ m)$$

In this formula, m is the modulus, a is the multiplier, c is the increment, and Z_0 the starting value or seed[Ref. 13, page 425]. The problem occurs because U_2 is a function of U_1 as shown in the recursive relation above. This dependency can cause X_1 and X_2 to fall on a spiral in (X_1, X_2) space, because they are not independent,

identically distributed, random variates. Because of the possibility of this kind of restrictive dependency, the Box and Muller method should not be used when only a single stream of a linear congruential generator is available, but can be used if two $U(0,1)$ random variables from separate seeds are available.[Ref. 13, page 425, 491]

A second method for obtaining standard $N(0,1)$ random variates is known as the polar method. This method is suitable for use with a single linear congruential generator seed. $N(0,1)$ random variates are generated using the following algorithm [Ref. 13, pages 491-492].

1. Generate U_1 and U_2 as IID $U(0,1)$ variables.

2. Let $V_i = 2U_i - 1$ for $i = 1, 2$.

3. Let $W = V_1^2 + V_2^2$.

4. If $W > 1$, go back to step 1.

5. If $W \leq 1$,
   let $Y = \sqrt{\frac{-2 \ln W}{W}}$
   let $X_1 = V_1 Y$
   let $X_2 = V_2 Y$.

6. X_1 and X_2 are IID $N(0,1)$ random variates.

b. *Generating Exponential Random Variates*

The other distribution needed for our SmartNet simulator was the exponential distribution. The exponential distribution is characterized by the first moment, sometimes called the mean or simply β. While the polar method is best suited for generating Gaussian random variates, the inverse-transform method proves to be the simplest and most accurate method for generating exponential variates. It is suitable because both the exponential distribution function and its inverse can be expressed using closed form equations. An exponential random variate X can be generated using the following simple algorithm [Ref. 13, page 486].

1. Generate U as an IID $U(0,1)$ variable.

2. Let $X = -\beta \ln U$.

3. Return X.

Figure 12 shows an exponential distribution with mean 100.
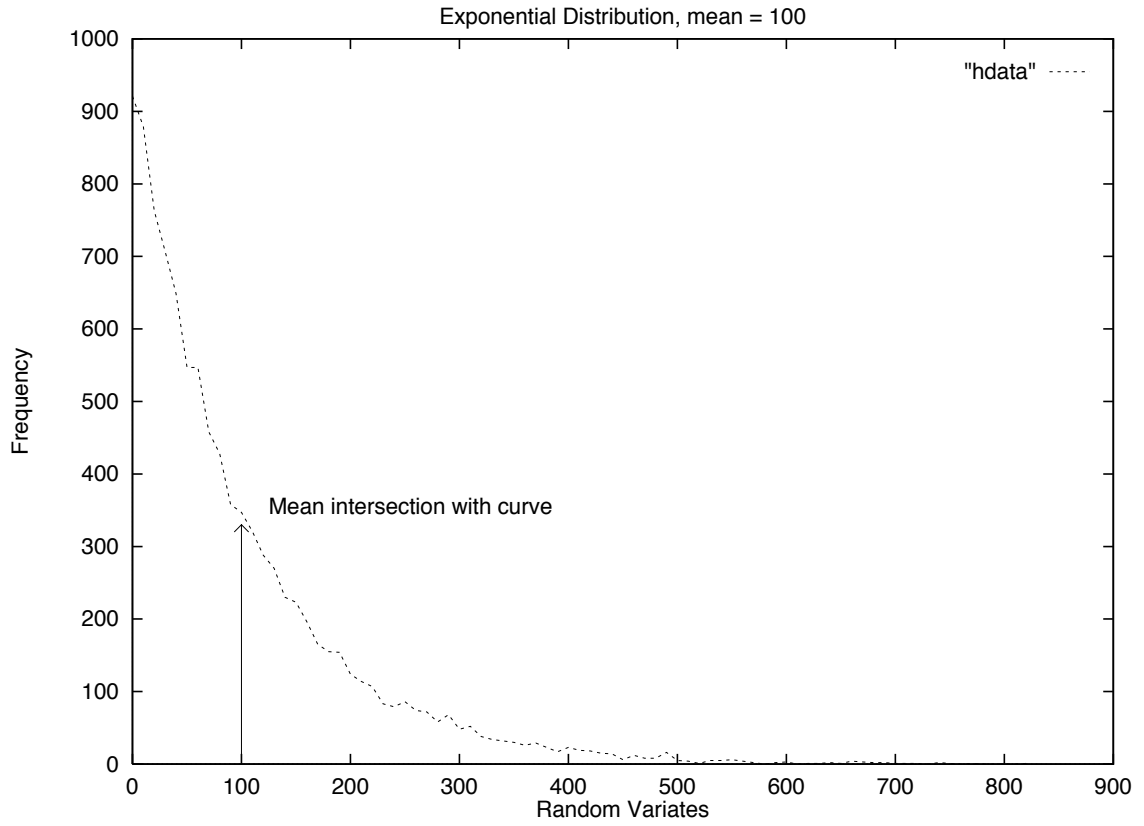
Figure 12. An Example of an Exponential Distribution, mean of 100.

E.   CONCLUDING REMARKS

This chapter has explained simulation in general, discrete event simulation in particular, and described in detail the generation of random variates for use in discrete event simulations. The next chapter will explain how discrete event simulation and random variate generation have been added directly to SmartNet [Ref. 1, 2, 3, 4].

IV.   THE SMARTNET SIMULATOR

A.   INTRODUCTION

This chapter explains changes and enhancements made to the original Smart-Net simulator[1] [Ref. 16]. The use of Discrete Event Simulation in the SmartNet simulator is discussed in Section C. Section D describes how we went about alleviating the limitations of the original SmartNet simulator.

B.   BACKGROUND INFORMATION

As we saw in Chapter II, SmartNet is a very capable scheduling framework with numerous and powerful operational modes. One of those modes is the SmartNet simulator mode. The simulator itself has powerful features that make it a useful tool; however, it also possesses certain limitations[2].

C.   DISCRETE EVENT SIMULATION AND THE SMART-NET SIMULATOR

The SmartNet simulator permits the operation of all aspects of SmartNet to be simulated using discrete event simulation. As we saw in Chapter III, when performing discrete event simulation, we need to identify events that trigger both the advancement of simulated time and the collection of system state variable data. Two of the events currently tracked by the SmartNet simulator are:

1. **Job Start:** This event occurs when a job is started (the actual execution of the job is simulated when SmartNet is run in simulator mode) on a machine in accordance with the schedule created by SmartNet.

2. **Job End:** This event occurs when job execution completes.

---

[1]The explanation of SmartNet provided in Chapter II provides more detailed definitions of many terms found in this chapter.

[2]Several of these limitations have been corrected via this research. Those changes are discussed within this chapter and in Appendices B and C.

These are two of the the most important events to SmartNet's run-time performance because they are the crucial components of the execution of the schedule that SmartNet creates[3]. These two events bracket a job's run-time, a duration that can take anywhere from micro-seconds to several days, depending upon the job and the machine. As a job begins, the time of its **Job Start** event is recorded and reported. When that same job completes (a **Job End** event), the run-time duration of that job is reported, and the simulation clock advanced to that point. In SmartNet simulation mode, the job does not actually execute, but a simulated run-time is used instead. The result is the ability to simulate the execution of a schedule that might take several days to run if the jobs were allowed to actually execute, but which takes several minutes instead. Figure 13 is an example demonstrating both the strength of discrete event simulation in SmartNet and illustrating event occurrences.

Unfortunately, we do not know what the exact run-time duration of a particular job on a particular machine would be. When SmartNet is actually running, start and finish times of jobs reflect actual wall clock time[4]. In this case, run-times are real. However, because the simulator does not actually execute jobs, an estimate of the actual run-time duration is needed.

1.   Advantages of the SmartNet Simulator

Using the SmartNet simulator provides definite advantages, both from the aspect of experimental capabilities and from the aspect of design. We already mentioned its capability to simulate the execution of complex schedules in several minutes that would, in reality, require days to complete. This capability gives SmartNet researchers the opportunity to compare the performance of different scheduling algorithms. Furthermore, there are design advantages because the simulator mode is built directly

---

[3]While the creation of a near-optimal schedule is the true benefit gained from using SmartNet, it is not an event on which we concentrate in our simulation experiments.

[4]Wall clock time is time as we perceive it throughout our day-to-day activities. It is the time we keep on the clocks in our home.

Real Time versus
Simulated Time Advance
in the SmartNet Simulator

Real Time 07:42:10
Job 2 Ends
Job 3 Begins
Simulated Time set to
00:12:20

Job 1   Job 2   Job 3

Event Occurs

Real Time 07:42:03
Job 1 Begins
Simulated Time set to
00:00

Real Time 07:42:07
Job 1 Ends
Job 2 Begins
Simulated Time set to
00:01:15

Real Time 07:42:12
Job 3 Ends
Simulated Time set to
00:20:30
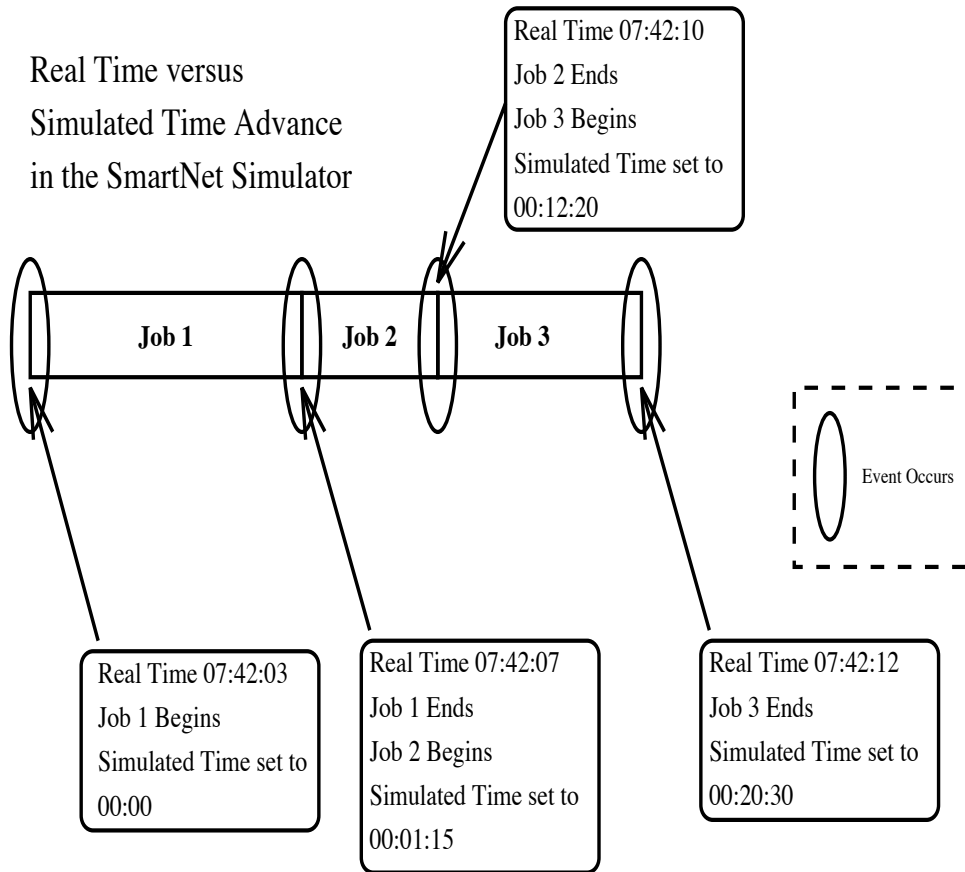
Figure 13. Real Time versus Simulated Time in the SmartNet Simulator. Three jobs, scheduled on one machine. The figure depicts simulated time advancement, real time, and event occurrences.

into SmartNet, helping the researcher to place a greater degree of confidence upon their research results. When using the SmartNet simulator, we are actually running SmartNet in *simulation mode*. This is important for two reasons. First, the simulator is an integral part of SmartNet, as opposed to being a removable segment of code or another application altogether. This means that the schedule, scheduling algorithms, database, default files, and inter- and intra-process communication resulting from or used by SmartNet in true operational mode are also used by SmartNet when run in simulation mode. Second, any and all changes to SmartNet source code, to include updates, implicitly change or update the simulator. There is no need for a duplication of effort, with one team working on improving SmartNet and another team working

49

on improving a simulation of SmartNet [Ref. 2]. We have an economy of effort that results in a better simulation tool.

2. Limitation of the Original SmartNet Simulator

The original SmartNet simulator had one major limitation. As we have seen, the simulator uses Expected Time to Complete (ETC) values for each job/machine pair, provided in the database, to build the schedule. Schedule-building is the intended use of the ETC values. *As a first attempt, the original simulator was built to use the ETC values found in the database as the simulated job run-time duration.* This meant that simulated jobs always ran for the exact amount of time they were scheduled to run. In reality, even when a job is the only load on all of the resources, the non-determinism associated with reading from/writing to disks and memory results in two different run-times for the same job with the same input. It is very difficult to exactly predict job run-times.

Therefore, our simulator should be able to simulate run-times of jobs according to run-time distribution characteristics found in various compute environments. We know that if a job is run repeatedly on a specific machine, it will almost never complete with the same duration. For example, if we run `JOB1` 1000 times on `MACHINE-A`, we may see 1000 different run-times. These 1000 run-time durations can be characterized by the distribution that they form. This distribution is specific to `JOB1` running on `MACHINE-A`[5]; `JOB1` running on `MACHINE-A` might always take *at least* 741.67 seconds to complete. The distribution of the completion times above 741.67 seconds might approximate an exponential distribution with mean 2.97.

---

[5]`JOB1` running on `MACHINE-B` may have an altogether different run-time distribution. This is particularly true if `MACHINE-B` and `MACHINE-A` are machines with different architectures or with different processing capabilities.

D. ALLEVIATING THE SMARTNET SIMULATOR LIM-ITATION

The SmartNet simulator needed to be modified so that the scheduled jobs that it simulates do not always execute for exactly the mean run-time. Specifically, we needed to alter the simulator so that run-time durations are not always identical to the ETC values used to create the schedule. The simulated run-time durations need to vary; however, they need to vary *realistically*. This should be done by incorporating run-time distribution data into the generation of simulated run-times. We have made these changes; they are presented in the following section.

1. Enhancements Made to the SmartNet Simulator

We enhanced the SmartNet simulator to allow job run-times to be derived from a run-time distribution. Doing so allowed jobs to be run with durations that varied in a well-defined way and was not always equal to the ETC values. The ETC values are either the mean of historical run-time durations or user estimates. Permitting jobs to run for non-ETC times entailed changes to both the simulator itself as well as to the I/O routines that read and write the SmartNet database. We added the ability to specify, within the database, not only a job's mean run-time, but also its type of distribution (recognizing both Gaussian and exponential distributions for reasons explained later) and both its second and third moments.

Due to the modular fashion in which SmartNet is built, the number of changes that we had to make to the actual code, above and beyond adding our own libraries, were few. However, we did spend a substantial amount of time reading the SmartNet code, identifying and fixing bugs, and correcting its Makefiles to operate correctly at our site [Ref. 17]. Appendices B and C provide detailed explanations of the files that we altered and created. We also enumerate the changes that we made to to each file. In our explanations in Appendices B and C, we name the enhanced and added files relative to the `SOLARIS` directory, which is where the SmartNet source code is installed. We will assume that these files will be located in the `SOLARIS/src/sn/program/`

51

subdirectory, unless otherwise explicitly stated.

E.   CONCLUDING REMARKS

With our enhancements, we now have a simulator that gives more realistic performance than the original version. We can alter characteristics of the run-time distribution for any and all job-machine pairs. Further, we have the ability to add additional distribution types with relative ease, since the random number generators, distribution name, and 1st, 2nd, and 3^d moments are already included in the database during the simulation.

V. EXPERIMENTS

A. INTRODUCTION

This chapter explains the simulation experiments we performed on SmartNet using the SmartNet simulator. The initial goal of the simulation experiments was to determine whether using intelligent scheduling would be beneficial, even if the jobs that were scheduled did not run for exactly the amount of time that we expected. In particular, we were concerned about whether it would still be beneficial to use intelligent scheduling if one or several jobs run for a substantially different amount of time than expected. Because determining a perfect schedule is an NP-complete problem, SmartNet is a scheduling framework for heterogeneous high performance computing that contains many different (polynomial) scheduling heuristics [Ref. 1]. These include several $O(mn^2)$ Greedy Algorithms [1], an $O(mn)$ Fast Greedy Algorithm, an $O(mn)$ Limited Best Assignment (LBA) Algorithm, an $O(mn)$ Opportunistic Load Balancing (OLB) Algorithm, and a variable complexity genetic algorithm. SmartNet pioneered the use of intelligent schedulers that accounted for both the Expected Time to Complete (ETC) of a job on each different machine and the expected load on each machine. In our simulation experiments we use the $O(mn^2)$ Greedy Algorithms, the $O(mn)$ Fast Greedy Algorithm, the OLB Algorithm, and the LBA Algorithm. All of the algorithms, except the OLB Algorithm, use the ETC value to compute the schedule. The LBA Algorithm does not take into account the expected load on the machines. The primary reason for this study is because jobs rarely execute for exactly the ETC time, which in SmartNet's case is generally the average of previous run-times with the same compute characteristics [Ref. 5]. This difference between actual and predicted run-times often occurs because all of the compute characterist-

---

[1] If an administrator installs SmartNet so that it uses these Greedy algorithms, SmartNet computes schedules for each of three different Greedy based algorithms and implements the one whose predicted performance is the best.

ics [Ref. 5] are not known or enumerated by the designer of the users program, and because the time to access memory and/or a disk is stochastic and not deterministic. In those cases where one or more of the jobs being scheduled have run-times that could differ substantially from the expected time, we need to determine whether there is still an advantage to using an algorithm that makes use of expected run-times or whether a computationally simpler algorithm that does not require looking up ETC values, such as Opportunistic Load Balancing (OLB), might not yield equivalently good performance.

As we began investigating this problem, we noticed that, for different ETC matrices[2], the performance of the various algorithms differed drastically. Therefore, in addition to our originally planned study, we categorized certain types of heterogeneity and ran experiments for many of these categories.

We ran our experiments using the SmartNet simulator mode rather than actually executing jobs under SmartNet. The simulator mode both gave us greater control over the input parameters and allowed us to complete more experiments in a reasonable amount of time. We begin this chapter with an explanation of the parameters we varied in the experiments. These parameters include both the distributions and various categories of heterogeneity. In Section C, we describe the simulation experiments that we performed, present the data, and explain our results. Finally, we discuss the theoretical performance limits of the SmartNet scheduling algorithms, compare the performance of SmartNet's $O(mn^2)$ Greedy Algorithm with its $O(mn)$ Fast Greedy scheduling algorithm, investigate the dependence of the performance of SmartNet's various algorithms on the arrival order of job requests, and finally examine the performance of some of SmartNet's algorithms when the matrix representing the job-machine ETC values is of mixed heterogeneity.

---

[2]An ETC matrix represents estimated performance of all the different jobs on all the different available machines. A specific element of the matrix represents Expected Time to Complete of a specific job (row) on a specific machine (column).

B.   PARAMETERS

As we developed the simulation experiments performed for this thesis, we found a need to specify two sets of parameters per experiment:

1. The run-time distributions used, and

2. the category of heterogeneity involved.

In order to determine some realistic job/machine run-time distributions that we would input into the SmartNet simulator for our experiments, we executed some programs on various parallel processors a statistically significant number of times and analyzed their run-time distributions. We describe these experiments in detail in Section 1. We expound fully on our categorization of job/machine heterogeneity in Section 2.

1.   Job Run-time Distributions

In Chapter III, we explained why job-machine run-times are typically not constant, but rather vary according to some distribution. We also discussed how we enhanced the SmartNet simulator to generate simulated run-time durations from a specified distribution, thereby permitting the simulation to more accurately reflect the true behavior of jobs. Testing the performance of SmartNet when the run-times of jobs are drawn from a particular distribution is essential to this thesis; but first we had to determine some realistic distributions that we would use in our simulations. Therefore, we repeatedly executed some parallel and sequential programs, gathered run-time statistics, and analyzed them.

We performed several experiments using the NAS Benchmarks [Ref. 18]. The NAS Benchmarks were used to determine the types of run-time distributions that may be typical for at least some jobs on some machines. We needed to determine sample parameters for these run-time distributions so that they could be reproduced by the SmartNet simulator. We used distributions and parameters observed during these NAS Benchmark tests for the run-time distributions in our simulation experiments.

While performing these tests, we controlled the following environmental characteristics.

- Server location. We ran experiments where the executable and input data and the output generated were located on the executing machine, as well as experiments where all of this data was located on a shared file server.

- Network and server load. When the executable and data were obtained from a file server, we ran experiments where both the network and the file server were both heavily and lightly loaded.

- Uni- or Multiprocessor. We ran some experiments where the programs had been compiled and executed on only a single processor of our Silicon Graphics multiprocessor computers, and other experiments where the programs were compiled and executed on multiple processors of the same machines.

- Amount of memory. We ran the jobs on two different multiprocessor Silicon Graphics machines. They each contained substantially different amounts of memory. One, `caesar`, had 64 MBytes and the other, `elvis`, had 192 Mbytes.

- Processor speed. `caesar` has four 200 Mhz MIPS R4400 processors, while `elvis` has four 150 Mhz MIPS R4400 processors.

We utilized a Silicon Graphics (SGI) Challenge-L multiprocessor machine and a SGI Onyx multiprocessor machine (elvis) throughout these experiments. They both ran the same version of the IRIX64 operating system, version 6.2. We used two machines so that the performance characteristics and run-time distributions of the jobs run in these experiments would provide us with a bigger picture of job run-time characteristics. Table V summarizes the configurations of the machines `caesar` and `elvis`.

The jobs that we used throughout these experiments were from two sources: NASA's reference implementation for some of the NAS Benchmarks, and our own implementations of other NAS Benchmarks that met the NAS Benchmark criteria. Four of the tests use some version of the NAS Integer Sort (IS) Benchmark, implemented either in parallel on four processors, or in single processor mode. Two other tests used the NAS Embarrassingly Parallel (EP) Benchmark run on a single processor. We now explain our experiments and their results.

56

	caesar	elvis
Type Machine	SGI Challenge L	SGI Onyx
Processor Speed	200 MHz	150 MHz
Processor Type	MIPS R4400	MIPS R4400
Number of Processors	4	4
Amount of Memory	64 Mbytes	192 Mbytes
Secondary Unified Instruction/Data Cache	4 Mb	1 Mb

Table V. Configuration of SGI machines `caesar` and `elvis`.

a. *Integer Sort, Executed on Four Processors*

This experiment examined the run-time distribution of a version of the NAS Integer Sort Benchmark executed on four processors. We implemented the integer sort using a counting sort [Ref. 6, pages 175–178] algorithm. We used Silicon Graphic's light weight process (thread) support functions, including `mfork()`, to implement our version of this benchmark. Below, we provide pseudo-code for the counting sort.

The number of initial values to be sorted (`TOTAL_KEYS`), which range between 1 and `MAX_KEY`, are stored in the array `key_array`. The algorithm first counts how many of each of the different values between 1 and `MAX_KEY` there are, storing the count in the corresponding element of the array `count_array`. When the algorithm completes, `final_array` will contain the original values but in sorted order.

```
for i = 1 to MAX_KEY count_array[i] = 0

for j = 1 to TOTAL_KEYS
    count_array[key_array[j]] = count_array[key_array[j] + 1]
comment:  count_array[i] now contains
              the number of elements equal to i

for i = 2 to MAX_KEY
    count_array[i] = count_array[i] +count_array[i - 1]
comment:  count_array[i] now contains
              the number of elements less than or equal to i
```

57

```
for j = TOTAL_KEYS down to 1
    final_array[count_array[key_array[j]]] = key_array[j]
    count_array[key_array[j]] = count_array[key_array[j]] - 1
comment:  final_array now holds the sorted keys
```

The actual code that we executed on the SGIs in shown in Appendix D.

We ran this sort across a heavily loaded network, obtaining both the executable and data from a file server that was also heavily loaded. When run on `caesar`, the run-time distribution, for 100 executions, appears Gaussian. Figure 14 shows a histogram of this distribution. When run on `elvis`, the run-time distribution, again for 100 executions, appears exponential and is shown in Figure 15. We note that the truncation of the exponential distribution shown in Figure 15 occurs at approximately 3.0. That means that the sort had to run for *at least* 3.0 seconds before stopping. The distribution that we see very closely matches an exponential distribution with a mean of around 0.20, translated 3.0 seconds to the right. We expect that many jobs would have a distribution similar to this, because all jobs have to run at least some amount of time[3].

In these experiments, we also see that memory size, and so, the need to swap to local disk, can have a definite effect upon the run-time distribution of a job. The integer sort on `elvis` completes, on average, 30% sooner than the same job on `caesar`. We note that, in this case, the amount of memory has more influence on the run-time of the job than does the speed of the processor. Of primary importance, however, is the observation indicating that the same job, running on two different machines, not only has different mean run-times, but the distribution of run-times is different, yielding a Gaussian-like distribution on one machine and an exponential-like distribution on the other.

---

[3]An exponential distribution is truncated at 0.0. If applied, without translation, in this case, that would mean there is the possibility of near-zero run-time.
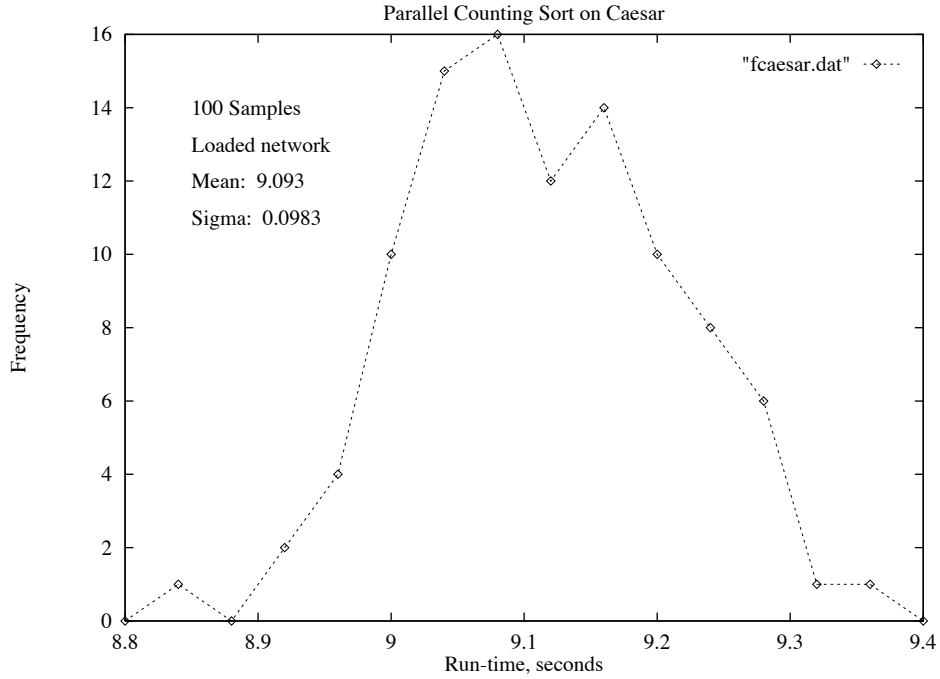
Figure 14. Forked Counting Sort, `caesar`.

b.　　*Integer Sort, Single Processor*

This experiment is the same as that discussed in the last section, with the exception of being run on a single processor instead of being distributed across four processors. Although a slightly different C++ implementation was used, see Appendix D, we again based our program on the counting sort pseudo-code presented earlier.

When the integer sort was run on `caesar`, the run-time distribution was not easily characterized; however, it appears related to a Gaussian distribution. The histogram of the distribution, shown in Figure 16, is multi-modal, which indicates that multiple distributions may be present. While this experiment does not provide us with definitive results, it does point to the fact that run-time distributions can be quite complex.

When the sequential integer sort was run on `elvis`, the run-time distributions were also multi-modal. Figure 17 shows a histogram of this run-time dis-

59
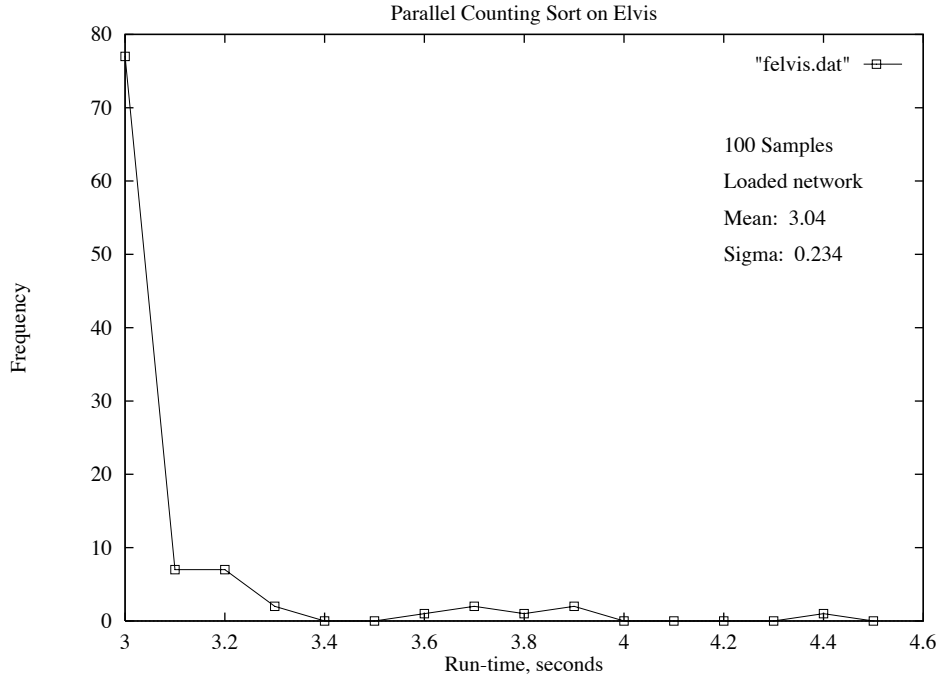
Figure 15. Forked Counting Sort, `elvis`.

tribution, which is also not easy to characterize. The multiple modes again suggests two different distributions which exist under perhaps different run-time specific conditions. We suspect that these conditions are related to changes in the network and server loads.

Once again, this set of experiments showed us that additional memory can greatly enhance run-time performance. The tests on `elvis` ran 700% faster than those tests run on `caesar`, which has the faster processors. The tests also show that run-time distributions can be very complex, and may be difficult to reproduce in a simulation. Although this thesis' experiments did not use such complex distributions, they should be modeled in future work.

c.   ***Embarrassingly Parallel NAS Benchmark***

The next set of experiments that we describe compared the run-time distributions of compute intensive jobs run from local disk to those run across the network from a file server. The tests that we describe in this section were executed
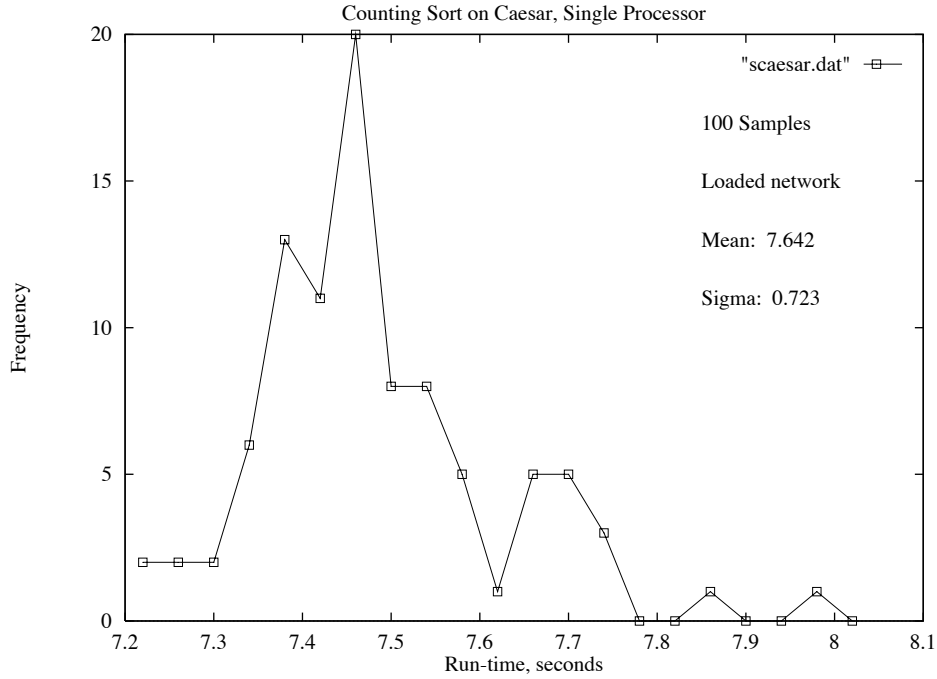
Figure 16. Counting Sort, `caesar`, single processor.

only on `caesar` because `elvis` did not have sufficiently large local disk available. We
used the reference implementation [Ref. 18], from NASA, of the NAS Embarrassingly
Parallel (EP) Benchmark. This implementation uses the portable message passing
interface(MPI) [Ref. 19] to parallelize the code. The tests we ran, however, were
compiled to be executed on a single processor[4]. The EP Benchmark was run 100
times for each test.

Figure 18 shows the run-time distribution of the EP Benchmark run
100 times when the executable is stored on `caesar`'s local disk. This distribution
appears exponential. We see the same effect here as we saw in the integer sort run on
four processors[5]. There is a shift of 741 seconds to the right, after which we see an
exponential distribution with mean 2.72.

---

[4]The MPI mechanism is still utilized in the EP Benchmark when it is compiled for a single
processor.

[5]The number of samples at the far left end of the distribution are small enough when compared
to the total number of samples to be considered a statistical fluke. The data point is included for

61
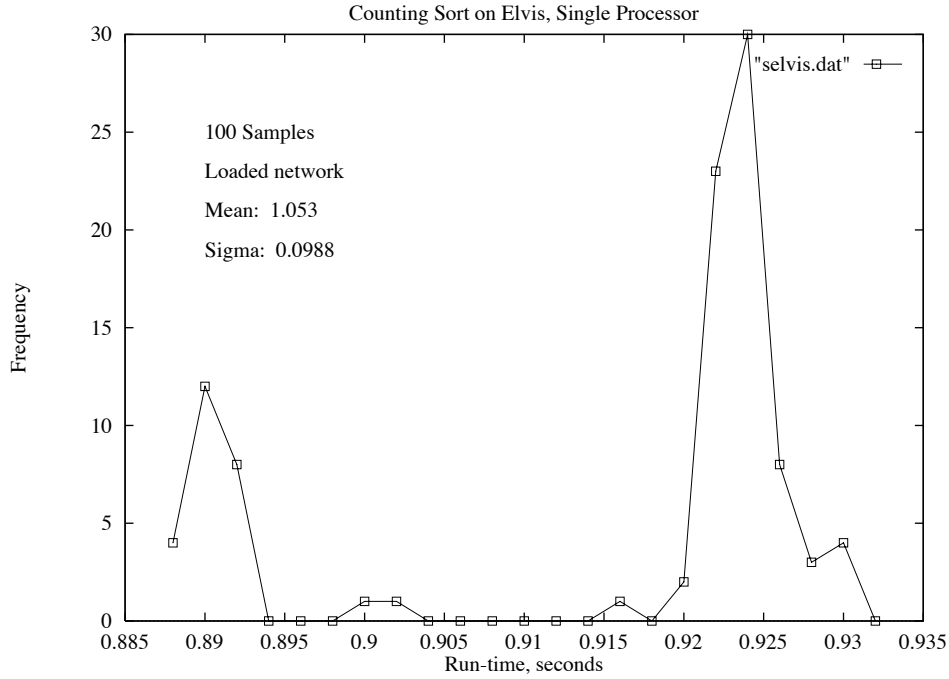
Figure 17. Counting Sort, `elvis`, single processor.

We also examined the run-time distribution of the same EP Benchmark code when executed on `caesar` but obtained across a lightly loaded network from a lightly loaded file server. Figure 19 shows the histogram from 100 EP Benchmark run-times. The run-time distribution appears to be truncated Gaussian[6]. Like the experiment above where the EP Benchmark was stored on local disk, the truncation value reflects the minimum time that it takes to run this EP Benchmark when the executable must be obtained from our particular file server over our local network. That truncation appears again at 741 seconds. The difference here, though, is that there is a different distribution of run-times throughout the range of values. We attribute this to the influence of other loads on the network and file server on the total compute time for reach job.

---

completeness.

[6]In this thesis, we sometimes use the term "truncated Gaussian" to refer to what is technically an Erlang or Gamma distribution. Both Erlang and Gamma distributions are strongly related to both Gaussian and exponential distributions.
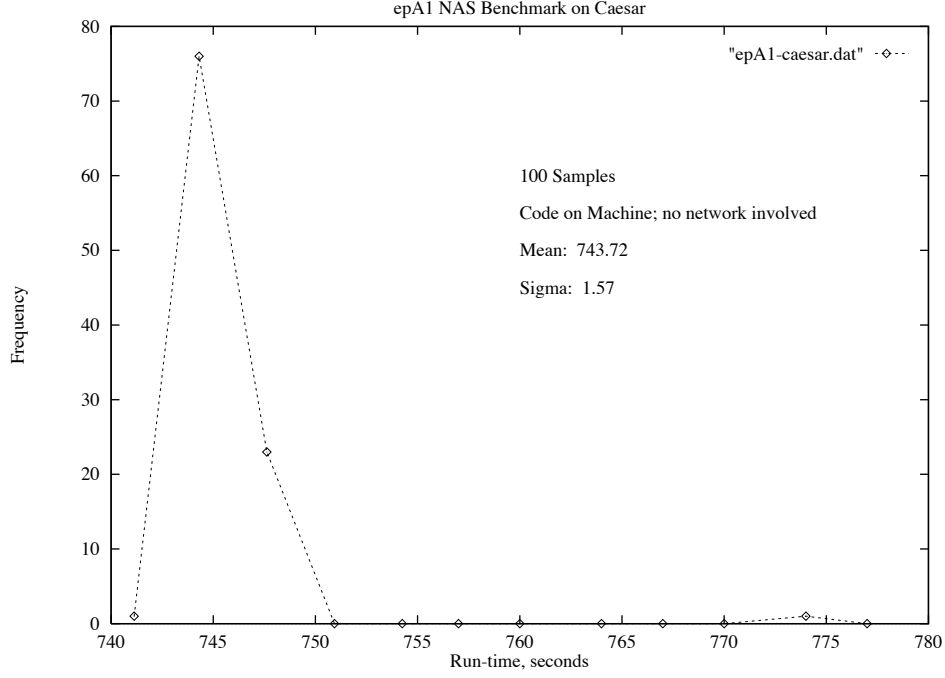
Figure 18. epA1 NAS Benchmark, Executable Residing on Local Disk.

2.   Categories of Heterogeneity

The other parameter that we need to examine and that we describe in this section concerns the category of heterogeneity we use in our experiments. We quantify the categories of heterogeneity according to two axes, one axis representing the job heterogeneity and the other axis representing machine heterogeneity. A heterogeneous computing environment is commonly thought of as a network of machines of differing or similar architectures, often having, at the very least, differing performance characteristics such as processor speed, quantities of cache, and amount of main memory. For example, two machines may be able to execute the same job, but one machine may execute that job an order of magnitude faster than the other machine. If the machines are nearly identical, then there is very little heterogeneity amongst the machines. If the machines are vastly different, then the collection of machines is very heterogeneous. Our categorization of heterogeneity encompasses this common-sense concept, but is more general in scope and more technically rigorous in its definition.
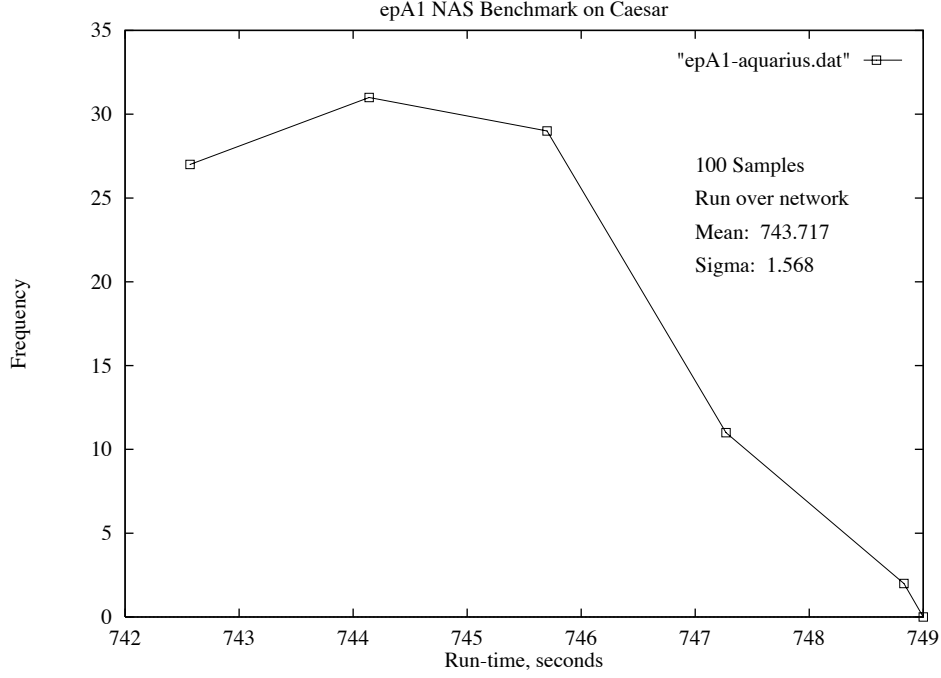
Figure 19. epA1 NAS Benchmark, Files obtained over a lightly loaded network.

However, both machines *and* jobs must be considered in any good characterization of computational heterogeneity. Jobs, like machines can be either very heterogeneous, slightly heterogeneous (e.g., one instantiation of a C++ compiler and another instantiation of the same C++ compiler executing with a higher specified level of optimization) or homogeneous (as we might expect to execute on special-purpose hardware). As an example, we consider a collection of jobs that is to be scheduled. If all the jobs are identical, e.g., all compiling the same source code and using the same specified run-time parameters, there is no heterogeneity amongst the jobs. If the jobs are all vastly different, then the jobs are very heterogeneous.

Therefore, we use two axes, one representing the heterogeneity of jobs and the other representing the heterogeneity of machines, to categorize the heterogeneity of a computing system. The relationship of job and machine heterogeneity is depicted in Figure 20, part (a).

We know that SmartNet uses estimates of the run-times of its different jobs
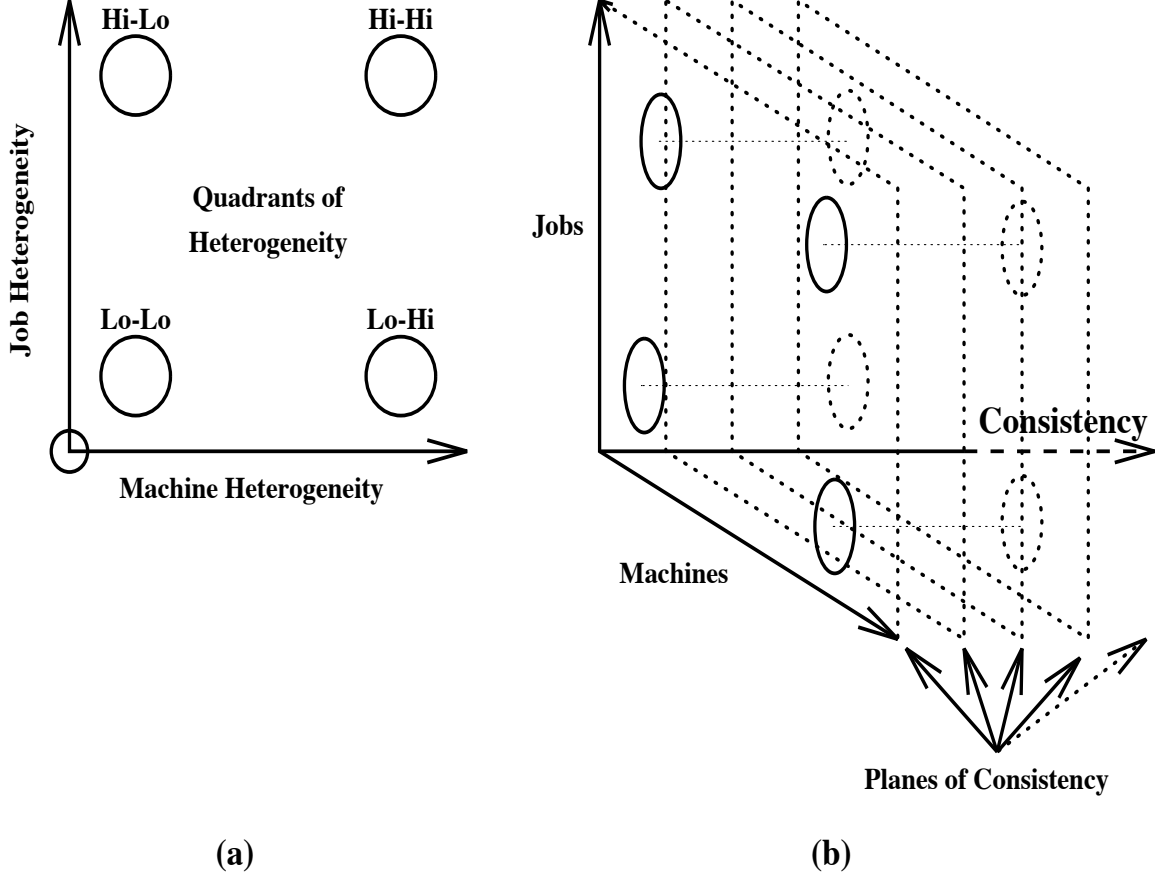
Figure 20. Quadrants of Heterogeneity and Categories of Consistency. Part (a) shows the two dimensional relationship of heterogeneity between jobs and machines. Part (b) shows the third dimension, consistency, and the numerous planes of consistency that can exist in different scenarios.

on its different machines to build a schedule detailing what jobs should run on which machine. For our simulation experiments, heterogeneity is introduced through setting appropriate parameters in the SmartNet database (See Appendix A). Specifically, heterogeneity of both jobs and machines is introduced into SmartNet through appropriately setting the ETC values of each job-machine combination present in the database. The actual database is quite complex, containing internet addresses of machines and (optionally) the longitudinal and latitudinal coordinates of those machines As such, we will represent its heterogeneity information in a more easily understood matrix format. An example of such a matrix is shown in Table VI.

	Machine					
Job		1	2	3	4	5
1	mean	30034	11	239	30097	533
2	mean	25	1003	8619	75	65037
3	mean	1078	93	1950	204001	8081
4	mean	35096	9501	29	2582	1000
5	mean	63	45055	1074075	11533	15
	Machine					
Job		6	7	8	9	10
1	mean	69	42799	1396	52453	4652
2	mean	30093	4723	11372	16333	287
3	mean	233	9	193	566	63526
4	mean	75019	23333	782	1134	1705
5	mean	403	207	6374	304291	666

Table VI. High-Job, High-Machine Heterogeneity Matrix.

For Table VI, we note that the average variance[7] for both the rows and the columns is very large, on the order of 10^{10}. Furthermore, we note that the distribution of both the column and row variances is unimodal. These facts indicate that the average job-machine run-times shown in this table fall at a point whose coordinates correspond to both High-Job Heterogeneity and High-Machine Heterogeneity (See Hi-Hi in Figure 20). In contrast, a matrix where the average variance for both the rows and the columns might be on the order of 10, would correspond to both lower machine and lower job heterogeneity (See Lo-Lo in Figure 20).

Our simulation experiments were built to examine four combinations of heterogeneity. It requires approximately 72 hours, not including setup time, to run a complete simulation experiment[8] and approximately six hours to run a Baseline

---

[7]The variances referred to here are variances of the run-time values in the ETC matrices.

[8]A complete simulation experiment requires that SmartNet build and execute 15 schedules for each database and the four different command files.

experiment[9]. We first chose to examine matrices representing four extreme values in our coordinate system. These four combinations can be thought of as *quadrants* of heterogeneity.

- High-Job, High-Machine Heterogeneity (Hi-Hi). All jobs perform very differently on all machines. As noted above, the variances for our complete matrix in Table VI, of both jobs and machines, are on the order of 10^{10}.

- High-Job, Low-Machine Heterogeneity (Hi-Lo). Each individual job performs similarly on all machines; however, no jobs perform similarly. For our sample matrix in Appendix E, the variance of jobs is on the order of 10^2, while the variance of machines is on the order of 10^8.

- Low-Job, High-Machine Heterogeneity (Lo-Hi). All jobs perform similarly on the same machine; however, the jobs obtain different performance on different machines. For our sample matrix in Appendix E, the variance the variance of jobs is on the order of 10^0, while the variance of machines is on the order of 10^7.

- Low-Job, Low-Machine Heterogeneity (Lo-Lo). All jobs perform similarly on every machine. For our sample matrix in Appendix E, the variance of both jobs and machines is on the order of 10^0.

There is a third dimension in the relationship between job and machine heterogeneity, however, which we call *consistency*. Consistency refers to the performance similarities of all jobs across machines. If all jobs perform best on the same machines (and subsequently perform worse on the same machines) then the schedule being executed is very consistent. We expect this situation to be common in some engineering laboratories where initially all machines might be workstations bought from the same manufacturer, with the same amount of memory and types of processor(s). As time goes on, machines are upgraded. A processor is added. Memory is added. But, typically, the machine with the fastest processor would also contain the most memory and the most cache. For now, we view this as adding a discrete axis to our already existing

---

[9]A Baseline experiment consists of SmartNet building and executing one schedule for a single database and each of the four different command files.

two axes of heterogeneity, one which represents just two, 2-dimensional planes: consistent and inconsistent. Future work is needed to determine how we might quantify this dimension as a continuous axis. Figure 21 shows the existence of consistency between two jobs and four machines. Conversely, if jobs perform well on different
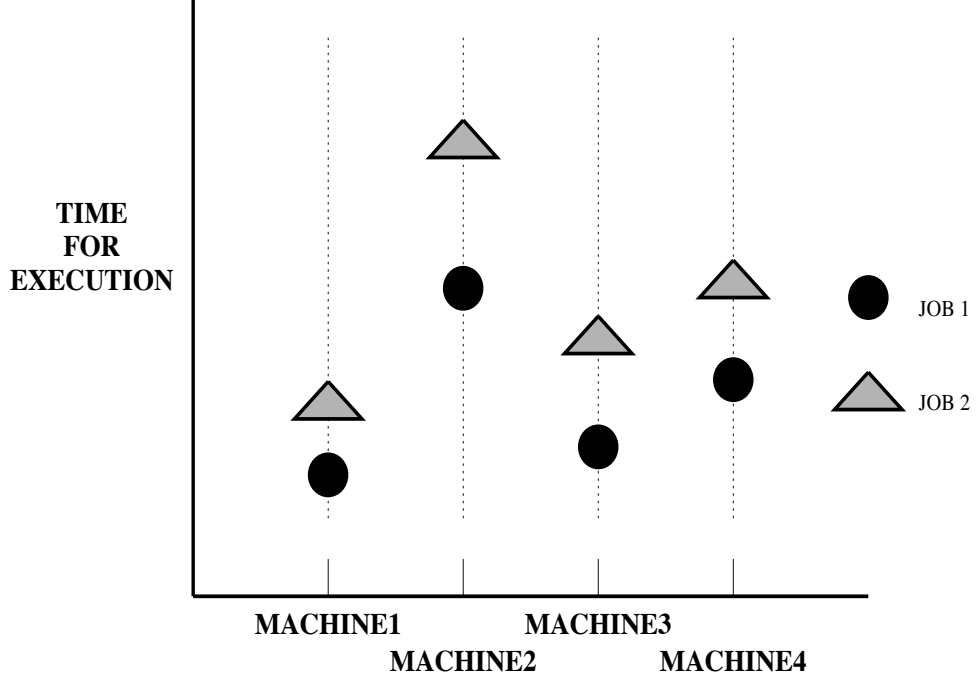
Figure 21. Consistency between two jobs and four machines. Both jobs perform better on the same machines.

machines, and poorly on different machines, then the schedule being executed is inconsistent. Figure 22 shows inconsistency between two jobs and four machines. We depict consistency, the third dimension of heterogeneity, in Figure 20, part (b).

To be brief, our nomenclature only includes mention of consistency if the matrix we are dealing with is consistent. In other words, when the term "High-Job, High-Machine Heterogeneity" is used, the matrix we are using is inconsistent. If the term "High-Job, High-Machine, Consistent Heterogeneity" is used, that matrix is consistent.

68
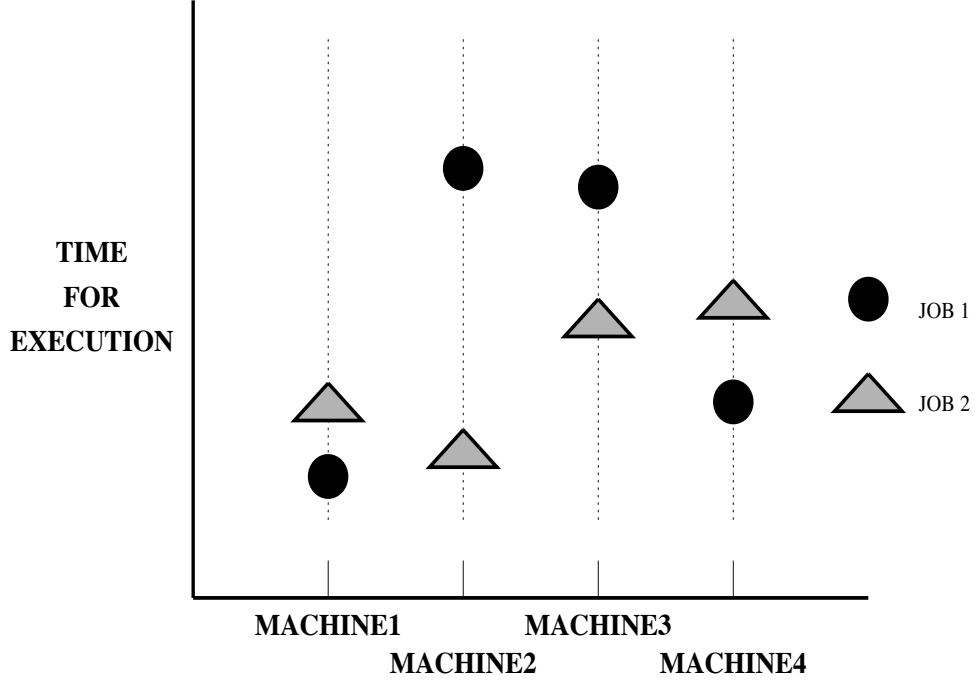
Figure 22. Inconsistency between two jobs and four machines. The jobs perform differently on the different machines; there is no consistency of performance.

C.   SIMULATION EXPERIMENTS

We performed two simulation experiments on SmartNet, aimed at examining how well the scheduling algorithms performed when the jobs scheduled did not execute for exactly the mean (of the previous run-times) specified in the SmartNet database. We first ran Baseline experiments that compared the performance of SmartNet's various algorithms for the different categories of heterogeneity, without considering consistency. Following that, we identified the Baseline matrices for which the $O(mn^2)$ Greedy Algorithm out-performed both the Opportunistic Load Balancing (OLB) Algorithm and the Limited Best Assignment (LBA) Algorithm. We term the matrices in this class to be **significant matrices**. We then ran experiments for consistent matrices that corresponded to the significant matrices, that is, we ran additional Baseline experiments using matrices that were identical to the significant matrices,

except that the contents of each row was sorted, from smallest to largest[10]. We term the sorted version of these matrices as consistent significant matrices. Finally, for all significant matrices, both consistent and inconsistent, we ran additional simulation experiments where the jobs did not execute for exactly the mean of the previous run-times; however, in one case the run-time distribution was assumed to be Gaussian, and for another case, it was assumed to be exponential. The details of the experiments are discussed in the following subsections.

Although the database (matrix) values for the experiments differed greatly, the conduct of the experiments was similar throughout. We now describe the features that were common to all of the experiments.

- Database Format. Although the job/machine heterogeneity differed for all databases created, each database contained mean run-times for each of five different jobs on each of ten different machines.

- Data Collection. Except for the Baseline experiments, all experiments in which the actual run-time of a job could differ from the predicted run-time of that job were executed 15 times. In each run, a different value was used to seed the random number generator that was used to generate the simulated "actual" run-time duration. The total time required to execute each schedule was summed and the average was computed. Multiple seeds were used to ensure that our results were not skewed[11]. We only ran the Baseline experiments one time, as the execution of this schedule was always the same (because jobs ran for exactly the predicted run-times).

- Scheduling Algorithms. We examined the performance of four scheduling algorithms, which are built into SmartNet, during each simulation experiment. These algorithms were explained in Chapter II and are listed below.

  - Opportunistic Load Balancing (OLB)
  - Limited Best Assignment (LBA)
  - Greedy, an $O(mn^2)$ algorithm

---

[10]We note that the average variance of each column is reduced by this sorting, but, as an example, for our High-Job, High-Machine Heterogeneity matrices, even the consistent matrices had an average column variance on the order of 10^{10}.

[11]This is a common method to reduce the influence of a single random number generation sequence that may be biased.

– Fast Greedy, an $O(mn)$ algorithm

Both the Greedy and Fast Greedy scheduling algorithms were pioneered by SmartNet. The LBA Algorithm is also contained in HeNCE [Ref. 20] and the OLB Algorithm is the only algorithm available in most resource management systems such as Condor, LoadLeveler, and NQE. SmartNet contains all of these algorithms, which are of different complexity, because SmartNet is a scheduling framework and different algorithms are appropriate for different environments. (See also previous work done by Benton and Lemanski on scheduling of network broadcasts [Ref. 9].)

- Job Request Format. When SmartNet is run in simulation mode, jobs are requested via a command file. The jobs can be requested either in groups or sequentially. For example, if we want to request `job4` to be executed three times, and `job5` to be executed 15 times, a grouped request would ask for `job4` to be run three times, and for `job5` to be run 15 times. To accomplish the same thing when jobs are submitted sequentially, we might request single executions of two different jobs in the order `job5`, `job4`, `job5`, `job4`, `job4`, and then 13 more single requests of `job5`. We looked at SmartNet scheduling algorithm performance when jobs were requested to be run in group format and randomly sequential format; however, the majority of our experiments were generated using randomized sequential requests. This was done because the order of job request affects the schedule. The Fast Greedy Algorithm maps and schedules the jobs on machines in the order in which they are submitted. The Greedy Algorithm uses the order to break ties. We chose to execute mostly singular requests both because they more closely mimic a real environment where different jobs are submitted by different users and because we wished to examine whether these algorithms performed better or worse when sequential, as opposed to grouped, requests were submitted.

- Job Request Sets. In order to ensure different results from the grouped method, we generated two random sequences of 125 job requests, which we will call `125-1` and `125-2`, where each individual request was chosen according to a uniform random distribution from among five different jobs. We also generated two more random sets, this time of 500 job requests, calling them `500-3` and `500-4`. We did this to look at performance variations between job request orderings, as well as to examine any performance differences that might occur because fewer or more jobs were requested.

- Actual Run-time Distributions. When we generated run-times that were different from the mean predicted run-times, we ran experiments for both Gaussian and exponential distributions.

Based upon our experiments with the NAS IS and EP Benchmarks above, we chose to implement a translated distribution with mean of 3.0 in our subsequent

simulation experiments. That is, we added the expected time to compute for a given job/machine pair, less the amount needed to keep the mean from changing, to a value drawn from an exponential distribution with mean of 3.0[12]. That is, the simulated run-times were generated using code represented by the following pseudo-code.

- X is the ETC of the job, available from the SmartNet database.
- $Y = X - 3.0$ (The 3.0 is taken from the experiments discussed previously.)
- Z is the random variate generated from an exponential distribution with mean 3.0.
- If ETC > 3.0, $Run\text{-}time\_duration = Y + Z$.
- If ETC ≤ 3.0, $Run\text{-}time\_duration = Z$.
- Return $Run\text{-}time\_duration$.

The actual code for this function is contained in Appendix C.

Again, based upon our earlier experiments described in Section 1, we chose to implement a truncated Gaussian distribution in our simulation experiments. We chose to truncate left of the mean at the mean less one sigma. Below is the pseudo-code for the algorithm we used to obtain a random variate from a truncated Gaussian distribution for run-time duration.

- $\sigma = \sqrt{2nd\_moment}$.
- while $Run\text{-}time\_duration > 1st\_moment - \sigma$
  * Generate random variate X from Gaussian distribution.
  * $Run\text{-}time\_duration = X$
- Return $Run\text{-}time\_duration$.

The pseudo-code describes code used in the function `generate_normal()`, which can be found in Appendix C.

1.   Baseline Experiments

These experiments were used to record SmartNet's performance when each job executed for exactly the amount of time for which it was scheduled. The Baseline experiment results show that there are circumstances where the Greedy and Fast Greedy Algorithms perform comparable to either OLB or LBA. Complete results

---

[12]In later experiments, we will also permit the mean for the exponential distribution to depend upon the job/machine pair.

from all of the Baseline experiments can be found in Appendix F. In this section, we provide graphical interpretations of typical SmartNet performance for a subset of the experiments. We note that if the run-time of an algorithm is not included in a graph below, it performed at least an order of magnitude worse than the included algorithms, and was omitted so that we could more readily distinguish between the remaining algorithms.

- High-Job, High-Machine Heterogeneity. See Figure 23. For the High-Job, High-Machine Heterogeneity matrix that we presented in Table VI, we see that Greedy and Fast Greedy perform comparable to LBA. Since LBA is a slightly less compute intensive scheduling algorithm, it may make sense to use the LBA scheduling algorithm instead of Greedy or Fast Greedy in such cases. The figure also shows how poorly the OLB Algorithm performs compared to the other three.
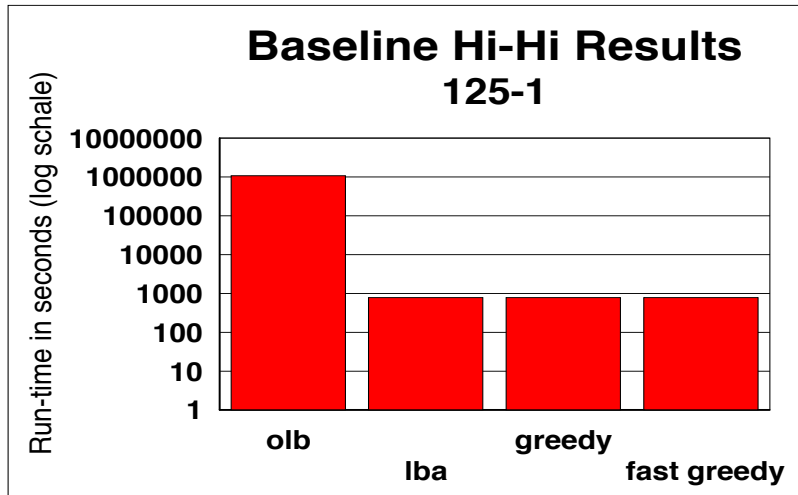
Figure 23.

- High-Job, Low-Machine Heterogeneity. See Figure 24. For our matrix chosen from the High-Job, Low-Machine Heterogeneity extreme, we saw that OLB performed just about as well as the Greedy and Fast Greedy Algorithms. OLB is also a computationally simpler scheduling algorithm. In this case, then, it may make sense to use the OLB scheduling algorithm instead of the Greedy or Fast Greedy Algorithms.

73
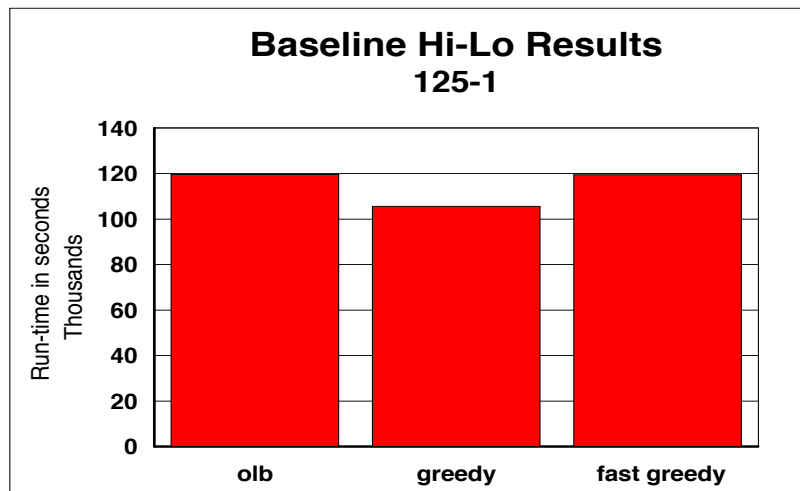
Figure 24.

- Low-Job, High-Machine Heterogeneity. See Figure 25. For our matrix chosen from the Low-Job, High-Machine Heterogeneity extreme, we saw that both the Greedy and the Fast Greedy Algorithms perform much better than OLB or LBA.
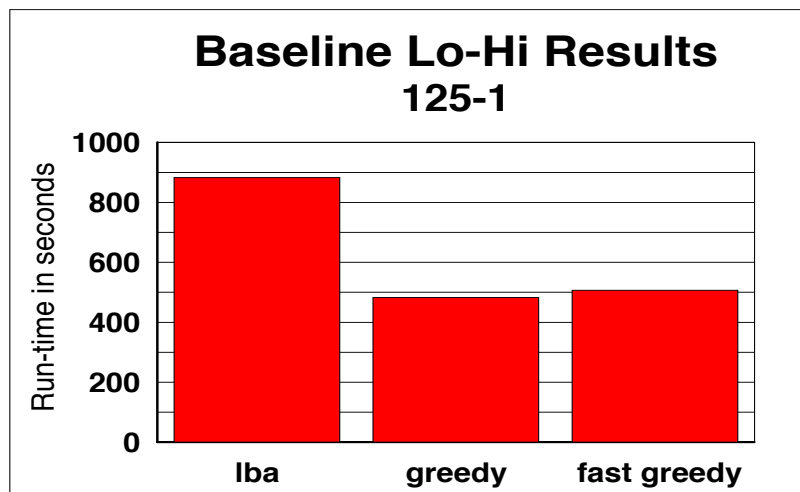
Figure 25.

- Low-Job, Low-Machine Heterogeneity. See Figure 26. For this matrix, both the Greedy and Fast Greedy Algorithms again perform comparable to OLB.
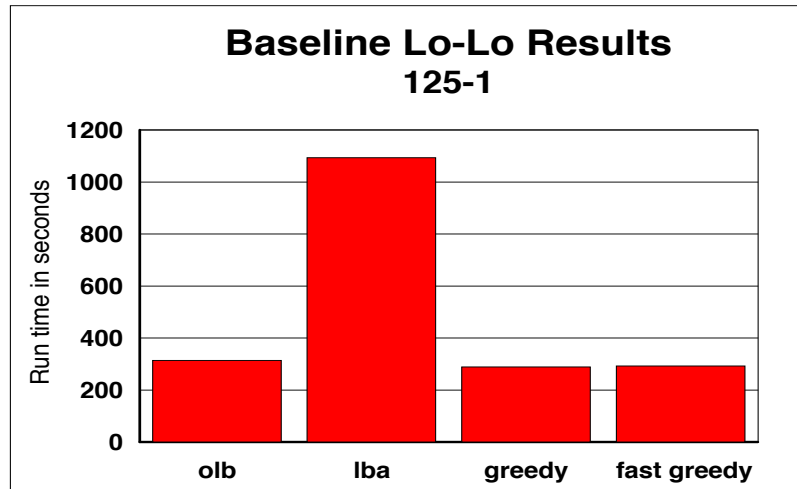
Figure 26.

We recall that consistency is the third dimension in the relationship between job and machine heterogeneity. We chose to examine two categories of heterogeneity along the consistency axis: High-Job, High-Machine, Consistent Heterogeneity, and Low-Job, High-Machine, Consistent Heterogeneity. These two categories are among some of the computing environments likely to be found today. When organizations purchase computers, they usually buy many similar machines. These machines get upgraded or replaced as money becomes available or as equipment breaks. Occasionally, more expensive, specialized computers are purchased in small numbers. These are added to the environment. This typically results in consistent behavior amongst machines — that is, there will be some machines that all jobs run well on, and some machines that all jobs run slower on. The results of the Baseline experiments implied that the most interesting run-time behavior would be found in the above two categories. We recognize that the other categories merit investigation, but are outside the scope of

this present thesis. We note that for both of these categories, the variance of the jobs and machines remains similar to that found in their inconsistent counterparts.

- High-Job, High-Machine, Consistent. See Figure 27. For our matrix chosen from this category of heterogeneity, Greedy and Fast Greedy perform better than either the OLB or LBA Algorithms.
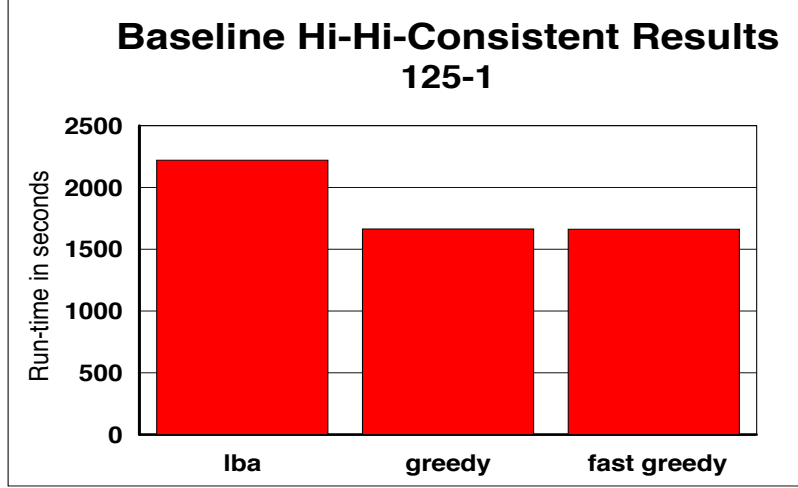
Figure 27.

- Low-Job, High-Machine, Consistent. See Figure 28. Again, for our matrix chosen from this category of heterogeneity, Greedy and Fast Greedy perform better than either the OLB or LBA Algorithms.

To briefly summarize the experiments we described above, we see, then, that from these six matrices, chosen from categories that represent the extreme ends of heterogeneity, the Greedy and Fast Greedy Algorithms develop schedules that are worthy of the extra compute time they required in three cases. Based upon these results, we opted to only further evaluate the Low-Job, High-Machine; High-Job, High-Machine, Consistent; and Low-Job, High-Machine, Consistent matrices in the remaining tests.

Baseline Lo-Hi-Consistent Results
125-1
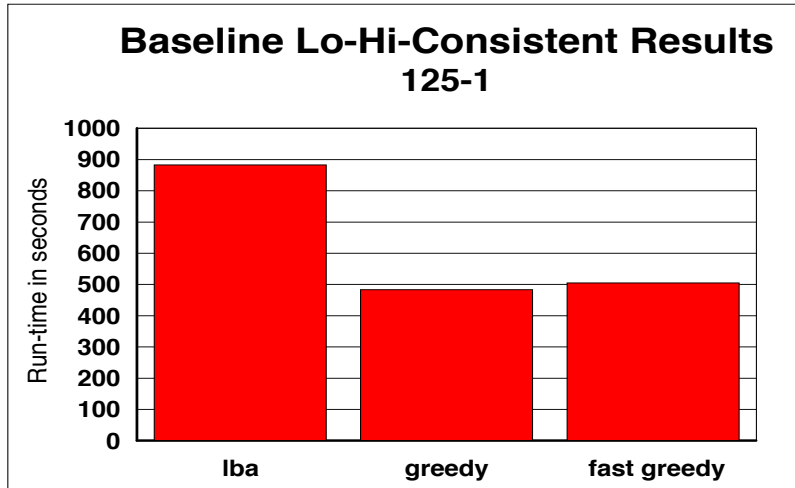
Figure 28.

2. Simulation Experiments where Jobs Ran for Times Different from the Predicted Run-times

This set of experiments examined the performance of the SmartNet scheduling algorithms when job run-times differed from the ETC values that were used to develop the schedule. For these tests, we used the enhancements that we made to the SmartNet simulator, described in Chapter IV. Using these enhancements, we were able to input the type of run-time distributions that the jobs being scheduled would have. Using the experiments described in Section B of this chapter, we determined the specific parameters needed to instantiate the distributions we might find in typical compute intensive jobs. We simulated jobs with both exponential and truncated Gaussian run-time distributions.

a. *Exponential Distribution Experiments*

The results of these experiments compare the performance of the various SmartNet scheduling algorithms when *all* jobs have an exponential run-time distribution. We recall from Section B of this chapter that the sample run-times from those experiments closely fit a shifted exponential distribution with mean 3.0. The

individual results from the exponential simulation experiments, which are consistent with the conclusions that we make in this section, can be found in Appendix F in Table XIX.

Figure 29.

Figure 30.

Exponential Lo-Hi-Consistent Results
500-4
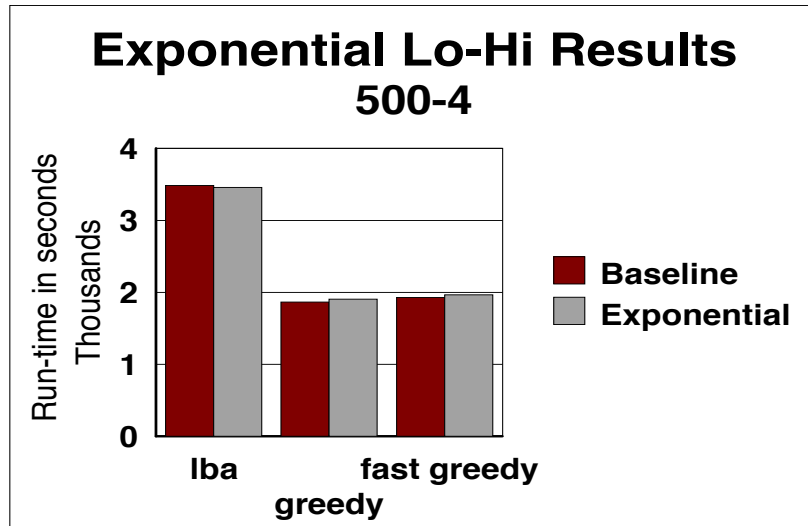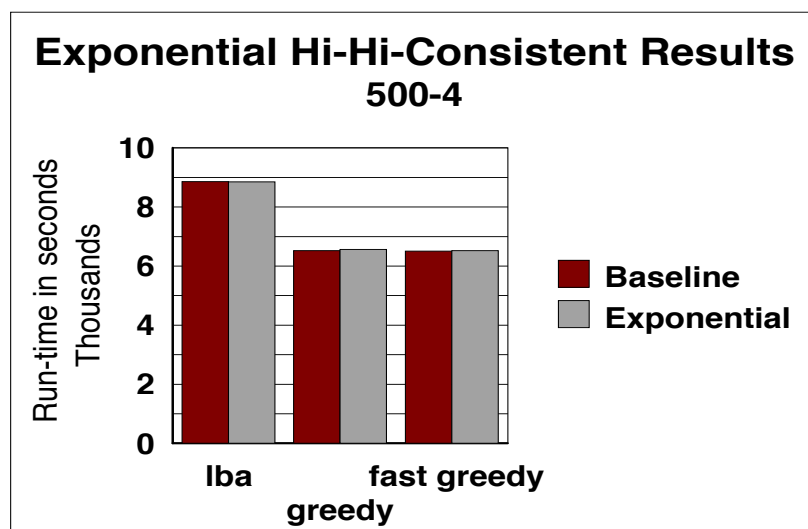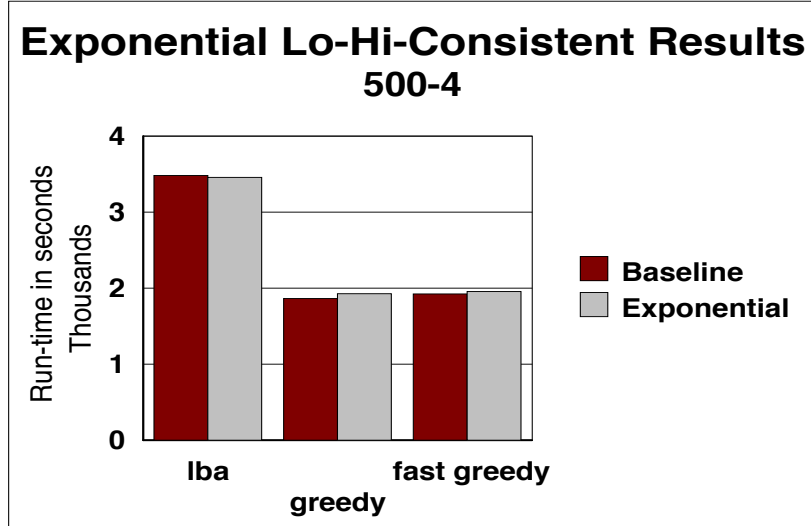
Figure 31.

When the results of these experiment are compared to the Baseline results, we see that jobs with exponential run-time distributions with mean 3.0 have completion times comparable to the Baseline results. Figures 29, 30, and 31 show these comparisons for the matrices we used in our simulations. These figures show that the schedules built by the SmartNet scheduling algorithms are still effective even though the actual run-time of a given job on a given machine can differ greatly from its corresponding ETC value.

b. *Truncated Gaussian Experiments*

These experiments were designed to examine the performance of the SmartNet scheduling algorithms when all jobs had truncated Gaussian run-time distributions. As in the previous experiment, this test takes advantage of the enhancements made to the SmartNet simulator. While the schedule was built using ETC data, the simulated run-times generated by the SmartNet simulator are taken from a truncated Gaussian distribution. In Section B, we discussed the characteristics of the truncated Gaussian run-time distribution characteristics obtained from running the NAS EP Benchmark. We determined from those experiments that truncation oc-

curred left of the mean at roughly $1st\_moment - \sqrt{2nd\_moment}$, or the mean less σ. Throughout this experiment, the mean, or $1st\_moment$, was the ETC value for the individual job/machine pairs, and the $2nd\_moment$ we set at 300% of the $1st\_moment$, or $3 \times mean$, to determine whether, if the variance was very large for all jobs, the Greedy and Fast Greedy Algorithms still performed much better than both the LBA and OLB algorithms. Any negative run-times that were generated occurred to the left of the truncation point, and so were not used in the experiments. The individual results from these experiments are included in Appendix F in Table XX.
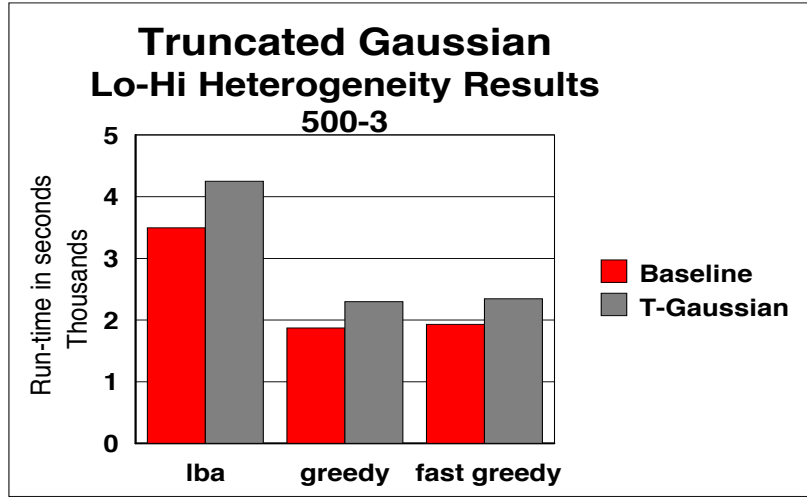
Figure 32.

The results in Figures 32, 33, and 34 show that the schedules are finishing up to 25% later than the schedules executed in the Baseline experiments. This is not unexpected, as truncation will shift the mean of the resulting distribution to the right. The results also show that the Greedy and Fast Greedy scheduling algorithms still perform better than the OLB and LBA Algorithms when job run-time distributions are truncated Gaussian with very large variances. Our experiments imply that is may be worthwhile to update the schedule as it is being executed to minimize the effect of the large job variances that result from run-time distributions with very large
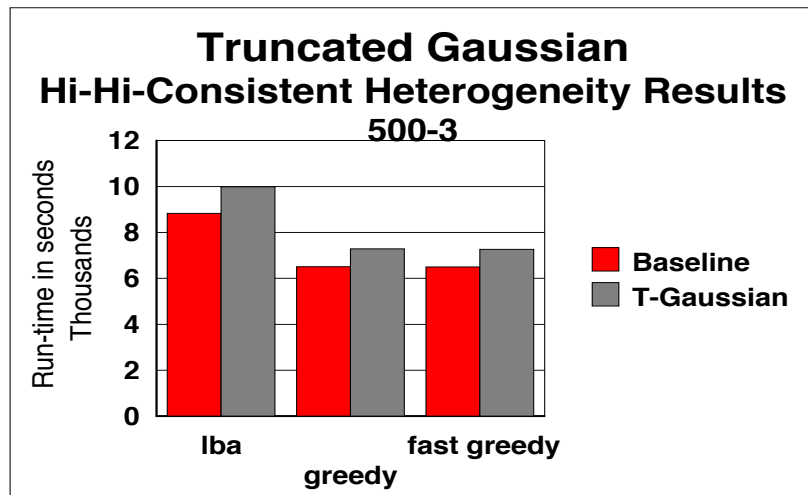
Figure 33.

variance, in this case, with variances of 300% of the mean. This claim is justified because preliminary evidence indicates that the observed 25% increase in the mean is not fully accounted for by the effects of truncation. This may warrant rescheduling because of its relatively low cost, especially for schedules involving many more
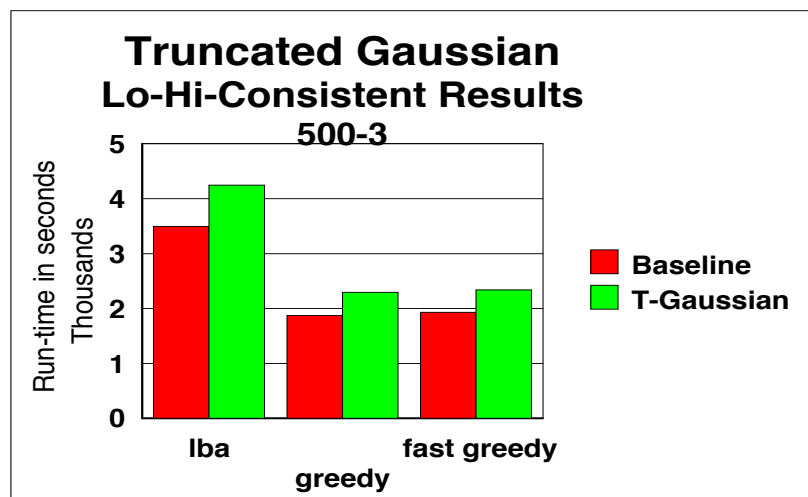
Figure 34.

machines and many more jobs than used throughout these simulation experiments.

D.   DISCUSSION

While performing the simulation experiments described previously in this chapter, we came across other aspects of SmartNet's performance that warranted examination. We first examine the performance of the SmartNet scheduling algorithms when compared to theoretical bounds. We follow that with a specific comparison of the Greedy and Fast Greedy Algorithms throughout all the simulation experiments. We then compare the performance the Greedy and Fast Greedy Algorithms when the jobs were submitted according to a uniform random distribution with the performance of those algorithms when the submitted requests are sorted and grouped according to job. Finally, we present another matrix with High-Job, High-Machine Heterogeneity characteristics, but which performs differently than expected.

1.   Theoretical Limits

SmartNet's Greedy and Fast Greedy scheduling algorithms consider both the time for each job to complete on each machine, as well as the current load on each machine when computing a schedule. Both Greedy and Fast Greedy compute near-optimal schedules in polynomial time. The NP-completeness of this scheduling problem and others, though, means that it would require exponential time to compute schedules that are optimal and that polynomial time schedulers can only approach this optimal. However, we are still interested in determining how close all the Baseline completion times are to the mathematical minimum. We now examine that issue for each of our six matrices that we enumerated above.

Assuming that we could examine one schedule every nanosecond, it would require more than 10^{93} years to determine, through exhaustion, which schedule would require the minimum amount of time to execute for one of our smallest experiments. For this reason, we instead use a less tight bound, though still a bound, that we now describe. We computed this bound, which we call the theoretical *Best Case Time*,

using the following method.

1. From the list of jobs submitted, determine how many of each job are being scheduled. This results in a *job count* for each job.

2. For each job, multiply the *job count* by the minimum amount of time that job could execute given that it was always assigned to its best machine, also assuming that no other type of job is assigned to that machine. This results in a *min group time* for each job.

3. Sum the *min group times*.

4. Divide the sum by the number of machines. The result is the Best Case Time for the schedule to execute.

For each matrix, we computed the *Best Case Time*, and compared that time to the Baseline time. The comprehensive results are shown in Table XXI, located in Appendix F. Table XXI shows us that we get closest to theoretical *Best Case Time*
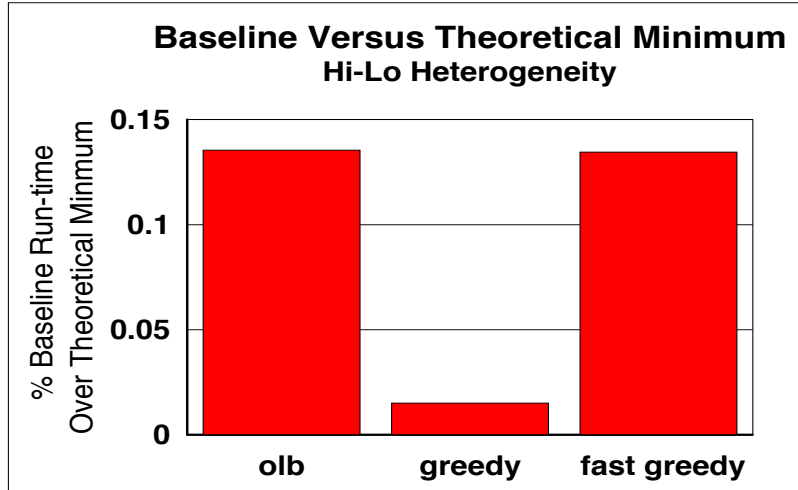
Figure 35. Theoretical Best versus Baseline Completion Time, High-Job, Low-Machine Heterogeneity. This data depicts the percentage difference between the theoretical *Best Case Time* and the Baseline completion time.

performance when schedules are created with our High-Job, Low-Machine and Low-Job, Low-Machine Heterogeneity databases. Figure 35 contains the High-Job, Low-Machine Heterogeneity comparison. Figure 36 contains the Low-Job, Low-Machine

Baseline Versus Theoretical Minimum
Lo-Lo Heterogeneity
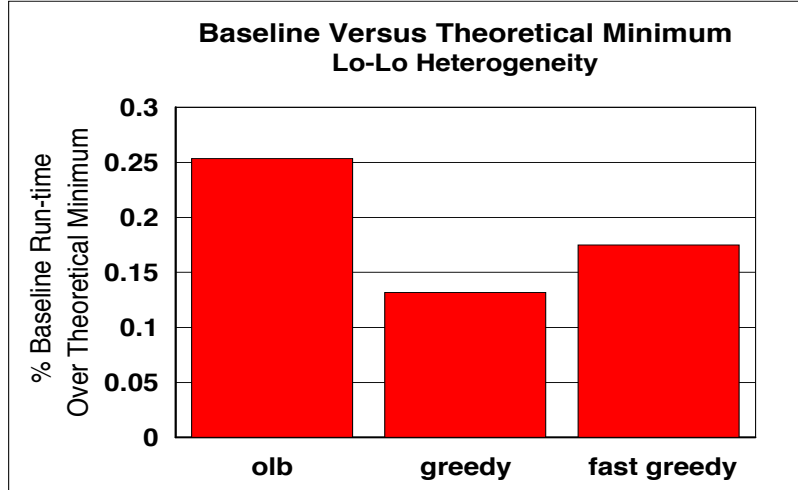
Figure 36. Theoretical Best versus Baseline Completion Time, Low-Job, Low-Machine Heterogeneity. This data depicts the percentage difference between the theoretical *Best Case Time* and the Baseline completion time.

Heterogeneity comparison. All of the other matrices show at least a 100% increase in run-time duration over the *Best Case Time*. This is because the machine heterogeneity is low, which means that the jobs all run fairly well on all machines. Low machine heterogeneity gives the algorithms more good choices of machines to schedules jobs upon. Whenever we have high machine heterogeneity, there are fewer near optimal machine choices for the jobs, and some jobs have to be run on machines that they do not perform well on. These results seem to indicate that the theoretical *Best Case Time* can be approached if the machines being utilized are very similar.

2. $O(mn)$ Fast Greedy versus $O(mn^2)$ Greedy

While performing the simulation experiments discussed previously, we saw the opportunity to compare the performance of two of the scheduling algorithms pioneered by SmartNet. The Greedy Algorithm has a complexity of $O(mn^2)$, while the Fast Greedy Algorithm has a complexity of $O(mn)$. What we wanted to know is how much of a performance gain we see when we invest in the more complex Greedy Algorithm.

This investment can be considerable for very large and complex schedules, and can have a significant effect upon overall SmartNet time of execution.
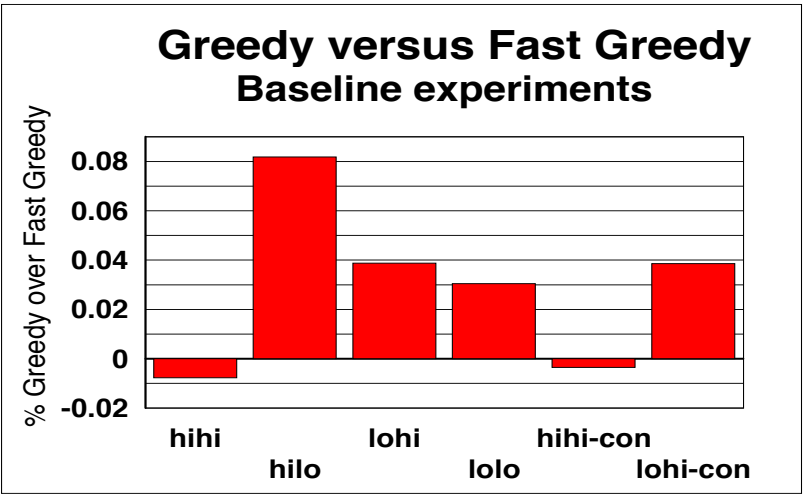
Figure 37.
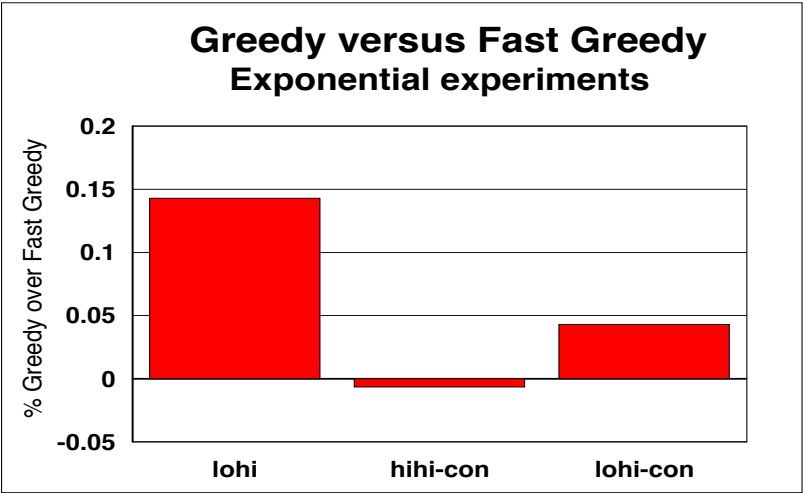
Figure 38.

Additional results are shown in Table XXII, located in Appendix F. Figures 37, 38, and 39 compare the performance of the Greedy to the Fast Greedy Algorithm
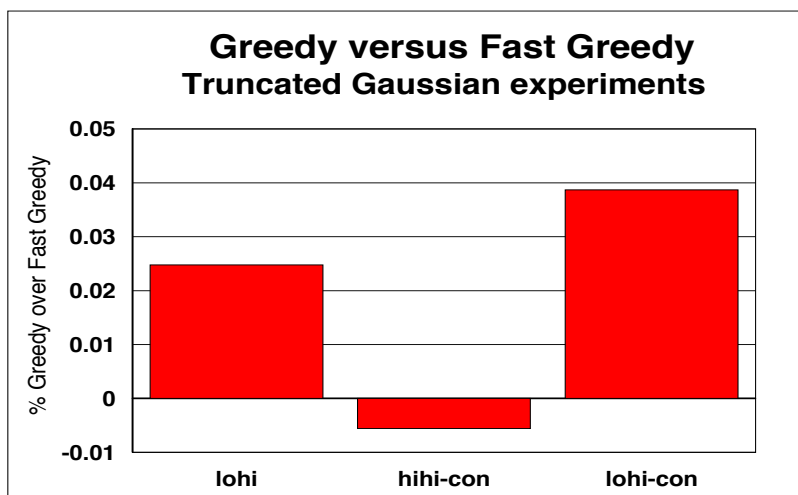
Figure 39.

for the the Baseline, exponential, and truncated Gaussian experiments. We averaged these run-times across all four sets of jobs. Figure 37 shows that Greedy schedules complete faster than Fast Greedy schedules for the High-Job, Low-Machine; Low-Job, High-Machine; Low-Job, Low-Machine; and Low-Job, High-Machine, Consistent categories of heterogeneity, but that Fast Greedy schedules complete faster for High-Job, High-Machine; and High-Job, High-Machine, Consistent categories of heterogeneity. Figure 38 shows that, for our experiments, when Greedy outperforms Fast Greedy, the gain is never more than 15%. What this tells us is that the better schedule execution time gained by using the $O(mn^2)$ Greedy Algorithm may not be worth the extra computational effort. Depending upon the time required to develop a schedule with the Greedy Algorithm, it may be more economical[13] to use the Fast Greedy scheduling algorithm. This thesis does not attempt to resolve that issue, as additional but related research needs to be performed that examines the completion times of schedules built using the two algorithms under many other different categories of heterogeneity. The question that needs to be answered is: Does a minimum of 15% decrease in schedule

---

[13]Economical from the standpoint of compute time required to build a schedule.

execution time warrant the use of a $O(mn^2)$ algorithm over a $O(mn)$ algorithm?

3.   Grouped Submissions versus Uniformly Distributed, Sequential Submissions

Earlier in this chapter, we discussed the method we used throughout our simulation experiments to request jobs to be run by SmartNet in simulation mode. We requested one of five different jobs, one at a time, repeatedly, via a command file, for a total of either 125 or 500 jobs. The jobs that were requested were chosen independently from a uniform distribution, so we call this method of choosing jobs the Sequential Method. We also described another method of requesting jobs, which we call the Grouped Method. Using the Grouped Method, jobs are requested in groups via the command file. `Job1` could be requested to run 25 times, which would be equivalent to requesting `Job1` to run once, but list that request 25 times in a row in the command file. During the course of our experiments, we became interested in knowing how schedules performed when jobs were requested with the Grouped Method as compared to their being requested in a random order using the Sequential Method. Specifically, we compared the performance of the Greedy Algorithm against the Fast Greedy Algorithm. We also varied, in other ways, the order in which the grouped jobs were requested in the command file, as we thought that may make a difference. We set up four command files, discussed below. In all cases, each request was chosen from the same group of 5 jobs.

- 125-up: 125 jobs requested in increasing order `job1` through `job5`, 25 repetitions of each job.

- 125-down: 125 jobs requested in decreasing order `job5` through `job1`, 25 repetitions of each job.

- 500-up: 500 jobs requested in increasing order `job1` through `job5`, 100 repetitions of each job.

- 500-down: 500 jobs requested in decreasing order `job5` through `job1`, 100 repetitions of each job.

87

Figure 40 shows how much faster Greedy schedules executed than the Fast Greedy schedules when using the Grouped Method of job requests. As before, a positive percentage means that the Greedy schedule executes faster than the Fast Greedy schedule.
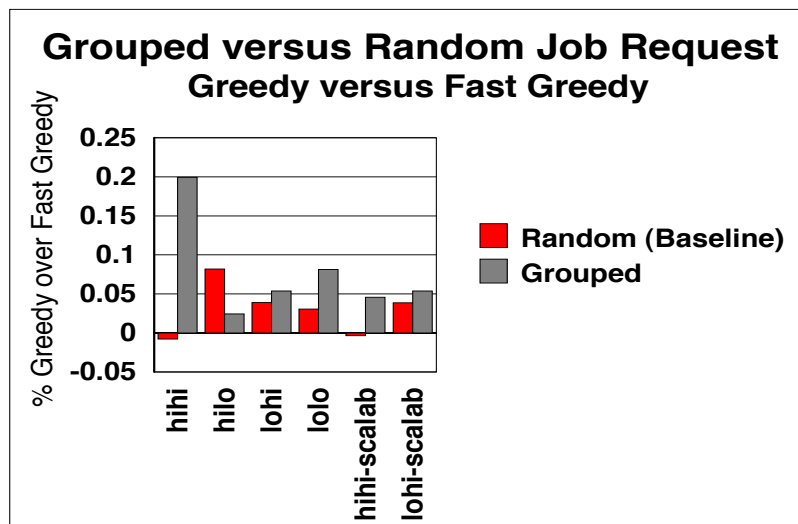
Figure 40. Greedy versus Fast Greedy, Grouped Method. This figure shows how much faster schedules built by the Greedy Algorithm finish executing versus schedules built by the Fast Greedy Algorithm. Positive values mean that the Greedy schedule is executed faster than the Fast Greedy schedule.

The results shown in Figure XXIII show significant differences between the two job request methods. The Sequential Method has Fast Greedy schedules completing before Greedy schedules under High-Job, High-Machine Heterogeneity; however, the Grouped method has Greedy schedule executing almost 20% faster than the Fast Greedy schedule. We see a similar contradiction for High-Job, High-Machine, Consistent.

Figure 41 shows that the performance of the Greedy Algorithm was not affected by the way that jobs were requested. For both the Grouped and Sequential methods, Greedy performed about the same. Figure 42 shows that the performance of the Fast Greedy Algorithm was slightly affected by the order in which jobs were requested.
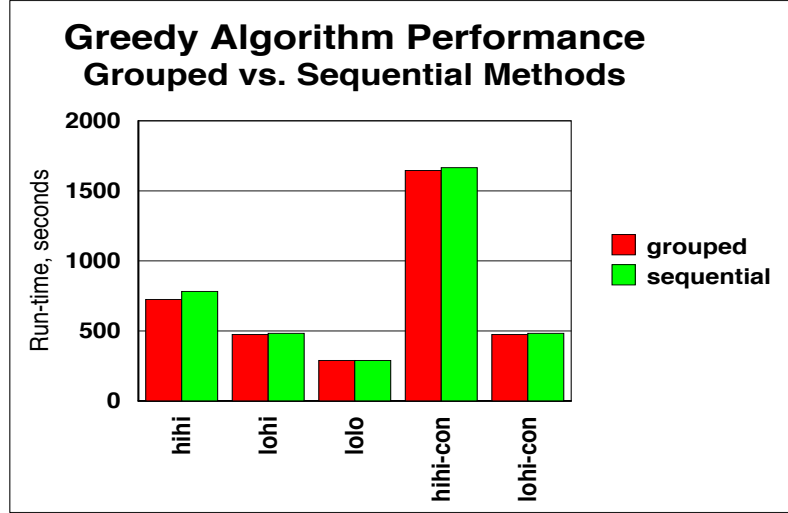
Figure 41. Greedy Performance; Grouped and Sequential Methods. Greedy performed about the same for both the Grouped and Sequential Methods.

4. Mixed Heterogeneity Matrices

Previously in this chapter we discussed the characteristics of High-Job, High-machine Heterogeneity. We noted that the distribution of the variances of the columns (machines) in the matrix was unimodal, and that the average variance for both rows and columns was on the order of 10^{10}.

We first thought that the magnitude of the variance was a simple way to characterize the category of heterogeneity. It turns out that this is not the best way to measure heterogeneity. We consider Table VII. Table VII includes row and column variances. The average row and column variance is on the order of 10^{10}. If we use only these variances, we might conclude that this matrix represented a High-Job, High-Machine Heterogeneity matrix. However, the last five machines are all very similar, and have a variance of 79.3. In fact, the distribution the column variances is bimodal. One mode is around 79.3, while the other is on the order of 10^{10}. What the matrix in Figure VII represents is a High-Job, High-Machine Heterogeneity matrix combined with a Low-Job, Low-Machine Heterogeneity matrix.
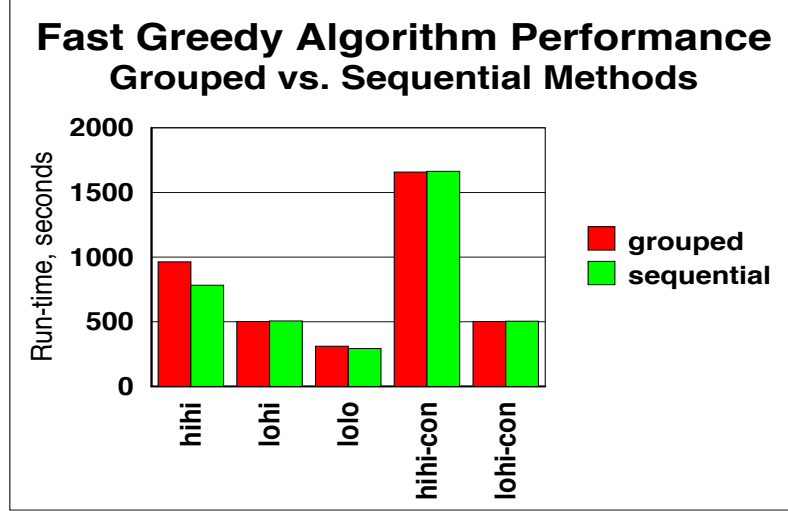
Figure 42. Fast Greedy Performance, Grouped and Sequential Methods. Fast Greedy performed slightly worse for both the Grouped and Sequential Methods.

When we ran our Baseline experiments on the Mixed Heterogeneity matrix in Table VII, we saw that both Greedy and Fast Greedy outperformed OLB and LBA by at least an order of magnitude. Recall that when we ran our Baseline experiments on our High-Job, High-Machine matrix, that Greedy and Fast Greedy performed similarly to LBA, while outperforming OLB. Also, when we ran the Baseline experiments on our Low-Job, Low-Machine Heterogeneity matrix, Greedy and Fast Greedy performed similarly to OLB.

These results show that row and column variance of a matrix are not suitable statistical characterizations of the categories of heterogeneity. In this thesis, we propose that the number of modes must also be considered. In this thesis, we primarily concentrate on matrices where both the row and column variances have only a single mode. Conclusions concerning other matrices, where either the row or column variances have multiple modes is beyond the scope of this thesis.

		Machine					
	Job	1	2	3	4	5	
1	mean	100,000	10	100	30000	500	
2	mean	25	1000	1200	75	65000	
3	mean	1000	33	1900	200000	8000	
4	mean	35000	9001	20	2500	100	
5	mean	50	15000	1000000	11500	15	
	variance	1.88×10^9	4.64×10^7	2.0×10^{11}	7.28×10^9	8.01×10^8	
		Machine					
	Job	6	7	8	9	10	variance
1	mean	11	12	13	14	15	$1,02 \times 10^9$
2	mean	26	27	28	29	30	4.19×10^8
3	mean	34	35	36	37	38	3.96×10^9
4	mean	21	22	23	24	25	1.22×10^8
5	mean	16	17	18	19	20	9.94×10^{10}
	variance	79.3	79.3	79.3	79.3	79.3	

Table VII. A Mixed Heterogeneity Matrix. The average row and column variance is on the order of 10^{10}.

E. CONCLUSION

This chapter has presented a considerable amount of detailed information about the experiments performed for this thesis. We explained the job distributions we chose to implement, as well as why we chose them. We also explained how we categorized heterogeneity. We presented our Baseline experiments and the results obtained, as well as the results from simulations where the jobs ran for times other than the predicted times. We examined how the Baseline results compared to the theoretical *Best Case Time*, and compared the performance of SmartNet's Greedy Algorithm to its Fast Greedy Algorithm, both when the job submissions were grouped as well as when they were individually submitted. We found that SmartNet embodies algorithms that performed well in all cases and began work on determining which of SmartNet's schedulers should be used for each of the various categories of heterogeneity.

VI.   SUMMARY AND FUTURE WORK

A.   SUMMARY

This thesis examined the effect of exponential and truncated Gaussian run-time distributions on the performance of SmartNet. In order to perform our experiments, we first had to enhance the original SmartNet simulator so that simulated job run-time durations could be non-deterministic. This non-deterministic behavior must be dictated by the type of run-time distribution that a specific job is designated as having. The result of this effort was a SmartNet simulator that behaves realistically within the bounds of the run-time distribution parameters we specified and implemented.

With our enhanced version of the SmartNet simulator, we were able to begin our examination of SmartNet performance. We discovered early in our experiments that we first had to determine the categories of heterogeneity that we wanted to examine. In addition, we needed a reference to which we could compare our results. These were our Baseline tests, which were tests of SmartNet designed such that the run-times did not differ from expected time to complete (ETC) values. The Baseline tests showed, for the specific categories of heterogeneity that were examined, the following results.

- For High-Job, High-Machine Heterogeneity (Inconsistent), SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms performed comparable to the LBA Algorithm, a slightly less complex algorithm than either Greedy or Fast Greedy. Because of this similarity of performance, we determined that further examination of Greedy and Fast Greedy scheduling algorithm performance was not needed for this category of heterogeneity.

- For High-Job, Low-Machine Heterogeneity (Inconsistent), SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms performed comparable to the OLB Algorithm, which is also a less complex algorithm than either Greedy or Fast Greedy. Additionally, OLB does not require the a priori information that is required by all of the Greedy algorithms (including Fast Greedy) and the LBA algorithm. The overhead of the Greedy and Fast Greedy scheduling algorithms is not warranted for this category of heterogeneity.

- For Low-Job, High-Machine Heterogeneity (Inconsistent), SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms performed significantly better than both OLB and LBA. We determined that further study of Greedy and Fast Greedy performance was warranted for this category of heterogeneity.

- For Low-Job, Low-Machine Heterogeneity (Inconsistent), SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms performed comparable once again to OLB. We determined that additional examination of Greedy and Fast Greedy scheduling algorithm performance was unwarranted for this category of heterogeneity.

- For High-Job, High-Machine Consistent Heterogeneity, SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms once again performed significantly better than both OLB and LBA. We again determined that further study of Greedy and Fast Greedy performance was warranted for this category of heterogeneity.

- For Low-Job, High-Machine Consistent Heterogeneity, SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms again performed significantly better than both OLB and LBA. We again, therefore, determined that further study of Greedy and Fast Greedy performance was warranted for this category of heterogeneity.

With our focus on Low-Job, High-Machine Heterogeneity; High-Job, High-Machine Consistent Heterogeneity; and Low-Job, High-Machine Consistent Heterogeneity; we began our experiments comparing the performance of the various SmartNet scheduling algorithms when jobs did not run for the length of time predicted. First, we examined the performance of SmartNet when the distribution underlying all jobs executed was exponential. The tests showed that the schedules built by the best SmartNet algorithms were still much better than those built by the less complex, non-intelligent SmartNet algorithms. Not only does this show that re-scheduling is often not needed after the initial schedule has been somewhat violated, but also that the overhead involved in using SmartNet's more intelligent algorithms is warranted even when the run-times of jobs can be significantly different from their predicted run-times.

We next examined the performance of SmartNet when the distribution underlying all jobs executed was a truncated Gaussian run-time distribution. The ETC values of the jobs were used as the *mean*, and truncation occurred to the left at $mean - \sigma$.

Variance for these tests was 300% of the *mean*. Our results show that SmartNet performance was somewhat affected by jobs whose run-times were from a truncated Gaussian run-time distribution. We saw up to a 25% increase in the time required to execute a schedule. Though much of this apparent decrease in performance was an artifact of our truncation method, some amount of it appears unaccounted for. This suggests that it may be necessary to recalculate a schedule for jobs that are still waiting to be executed when we have jobs with this run-time behavior. The relatively low cost of rescheduling may help minimize any resulting decrease in performance low. In this case, also, we see that the overhead involved in using SmartNet's more intelligent algorithms is warranted even when the actual run-times of jobs can be significantly different from their predicted run-times.

As we performed our experiments, we came across other related areas of Smart-Net performance that we were able to examine. First, we looked at the theoretical minimum execution time of a schedule and compared that theoretical minimum to the performance of the four scheduling algorithms we tested. Our results showed that SmartNet's algorithms often approach the theoretical limits when running tests with our High-Job, Low-Machine; and Low-Job, Low-Machine categories of heterogeneity. In all other cases, the algorithms performed at least 100% worse than the theoretical minimum. We therefore conclude that, for our test environment, SmartNet was able to build near optimal schedules when the variation in performance of jobs on machines was low.

Next, we compared the performance of SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms. We determined that the schedules built with the Greedy algorithm executed faster than those built with Fast Greedy for High-Job, Low-Machine Heterogeneity; Low-Job, High-Machine Heterogeneity; Low-Job, Low-Machine Heterogeneity; and Low-Job, High-Machine Consistent Heterogeneity. The performance gain was never more than 15%, however, when jobs were submitted in a random order. For all other categories of heterogeneity, schedules built by the

Fast Greedy scheduling algorithm completed faster than those built with Greedy. We determined that the cost to schedule with the more complex Greedy algorithm may not always outweigh the performance gain, and that such considerations needed to be further examined in future research.

We then compared the performance of SmartNet's intelligent schedulers when jobs were requested sequentially and randomly, which we called the Sequential Method, against the Grouped Method. Our results showed significant differences in the performance of the Greedy and Fast Greedy scheduling algorithms when these two methods were used. We conclude that there is a need for both methods to be used within SmartNet, but that they need to be used appropriately. Further, the differences in performance between these two job request methods needs to be accounted for when deciding which scheduling algorithm to use.

Lastly, we examined a Mixed Heterogeneity Matrix. While both the average row and column variance was on the order of 10^{10}, and so might have appeared to be a High-Job, High-Machine Heterogeneity matrix, a closer look at the distributions of the row and column variances showed us this matrix was very different. The distribution of the row and column variances for our first matrix was uni-modal, which we concluded was characteristic of the High-Job, High-Machine category of heterogeneity. However, the distribution of the column variances of the second matrix was bi-modal. We concluded that the existence of more than one mode meant that a matrix was actually a combination of two different matrices corresponding to two categories of heterogeneity — in this case, a High-Job, High-Machine matrix and a Low-Job, Low-Machine matrix. When we compared the results of the Baseline experiment for the Mixed Heterogeneity Matrix with our High-Job, High-Machine Heterogeneity matrix, we saw significant differences in the performance of the Greedy and Fast Greedy algorithms. These results helped us determine that categories of heterogeneity could not be statistically categorized by average row and column variance, but that additional statistical study was needed.

96

Overall, we determined that SmartNet's algorithms perform well under the categories of heterogeneity we identified, and that additional research is needed to further pinpoint ways to increase performance in the many different computing and network environments likely to be found in the Department of Defense.

B. FUTURE WORK

There are numerous opportunities for future work related to this thesis. First, SmartNet performance needs to be further evaluated using additional matrices from the categories of heterogeneity that we identified as well as with additional examples of matrices with Mixed Heterogeneity. Additionally, the categories of heterogeneity most often found in typical environments needs to be further researched. SmartNet performance needs to be further examined when the jobs' run-time distributions are different from the ones that we simulated. This creates a need for more study into what types of distributions we should expect to find in various high performance computing environments. Further, SmartNet performance should be evaluated when different jobs execute with different types of run-time distributions. The cost-effectiveness of SmartNet's $O(mn^2)$ Greedy and $O(mn)$ Fast Greedy scheduling algorithms needs to be traded off against their performance, and the cost and benefits of rescheduling should also remain a consideration.

APPENDIX A. SMARTNET DATABASE FORMAT

Tables VIII, IX, X, and XI outline the format of the SmartNet database. They include fields added because of research performed in this thesis.

Site Object Fields	Format
site name	search key string
description	string
latitiude	float, global coordinate
longitude	float, global coordinate
bandwidth	float, in bytes/second (within site)
latency	float, in seconds (within site)
notional	integer, 1 or 0 (true or false)
status	integer (unused at this time)

Table VIII. Site Object Database Format

Machine Object Fields	Format
machine name	search key string
architecture	string (unused at this time)
IP address	standard internet dot notation
description	string
location	string
relative cost	float (unused at this time)
relative performance rate	float (unused at this time)
Is the machine notional?	integer, 1 or 0 (true or false)
site name	string

Table IX. Machine Object Database Format

Model Object Field	Format
model name	search key string
description	string
idempotent	integer, 1 or 0 (true of false)
The number of compute characteristic description lines	integer
compute characteristic's descriptions, one line per description	string

Table X. Model Object Database Format

Model-Machine Object Fields	Format
machine name	search key string
model name	search key string
group name	search key string
distribution type	string (**ARMSTRONG ADDED**)
first moment	float (**ARMSTRONG ADDED**)
second monent	float (**ARMSTRONG ADDED**)
third moment	float (**ARMSTRONG ADDED**)
theoretical compute function	equation, producing seconds
theoretical network function	equation, producing seconds
theoretical data use function	equation, producing bytes
theoretical floating-point function	equation, producing Mflops
relative execution rate	float (unused at this time)
experiential compute data written to database by smartnet	
The number of compute characteristic description lines	integer
compute characteristic's descriptions, one line per description	string
experiential network data written to database by smartnet	

Table XI. Model-Machine Object Database Format

APPENDIX B. ENHANCEMENTS MADE TO EXISTING SMARTNET CODE

1.   INTRODUCTION

This appendix provides detailed explanations of the changes made to several SmartNet files. The changes were made in order to enhance the SmartNet simulator. Chapter IV provides an explanation as to why these changes were required.

2.   SERVER/SIMULATOR/JOBSTARTEVENT.CC

This file details the member functions of the `JobStartEvent` class. There are only two functions to this class: a constructor, and the function `execute()`. The `execute()` function does several things, but only one thing that we are interested in changing. The duration that a job is to run in simulation mode was retrieved from the ETC information provided in the input database. This is where the crux of the problem with the simulator lay. The duration retrieved is the exactly the same as the ETC value that the schedule was built from. The function call was:

- job duration = ETC of job provided in database

We changed the above call to:

- job duration = run-time of job calculated from distribution data

The distribution data is provided in the database file (another change). The function required to calculate the job run-time, based upon this distribution information, is an addition to the SmartNet simulator code.

101

3. SERVER/SN-LOG/SN-LOG.C

This file is the program code for the SmartNet `logger`, which listens to various SmartNet messages and logs specific detail to an output file. This output log file can then be used to recreate SmartNet runs using the SmartNet `replay` mechanism. In the case of the SmartNet simulator, the `logger` is used to capture run-time and for scheduling information for later evaluation of SmartNet's performance and behavior.

There were minor enhancements made to this file, but they were important. We found that the code was not outputting the correct time for the duration that a job was running in simulation mode. The same was true for the times recorded for jobs to begin. This stemmed from the use of the ETC value for both scheduling and running jobs in simulation mode. The changes made involved altering variable accesses in the following functions:

- `JobNoticeStart`: access true start time versus time variable t
- `JobUpdateDone`: report true finish time/duration vice time variable t

4. SN-SUBMIT/EXTERNAL.C

This file contains external interface code specific to the sn-submit program. The `sn-submit` program must be run to actually submit jobs to SmartNet via command line. Command line submission must be used in simulation mode, because SmartNet's graphical user interface does not support simulation mode.

While investigating necessary changes to the SmartNet code, it became evident that `sn-submit` was trying to actually start the schedule on the prescribed machines. This needed to be fixed in order for the simulator to actually be a simulation tool. We fixed this problem by checking to see whether simulation mode had been set when `smartnet-master` was started. The check for simulation mode was performed in the `sn_external_start()` function, and was performed as follows:

- If simulation mode is set, return true.

The change allowed sn-submit to run in simulation mode without attempting to actually start the schedule.

5.   SN-SUBMIT/SUBMIT.C

After examining and changing the sn-submit/external.C file, it became evident that we needed to be able to start sn-submit in simulation mode. The file `submit.C` contains the main program for the `sn-submit` application. We needed to add simulation functionality at the command line. Simulation functionality included being able to use ``-S´´ as a command line argument to `sn-submit`. It also included setting the simulation mode global variable to true. We added the equivalent of the following pseudo-code.

- Global Integer Variable `simulationMode` = false;

- If sn-submit includes -S as a command line argument,

    - Set simulationMode to true;
    - Remove `-S` from the input argument list;

6.   SN-SUBMIT/README

This file includes detailed information on how to run `sn-submit`. We changed the README file to include information about the ``-S´´ flag, thus informing the user how to run `sn-submit` in simulation mode.

7.   SERVER/SRC/MODELMACHINE.H

This is the header file for the `ModelMachine` class. The `ModelMachine` class handles all of the characteristics of individual job-machine pairs. Much of the data is provided via the input database file. The format of the database file is included in Appendix A.

Runtime distribution information is necessary for each individual model-machine pair. In order for the user to specify this information (for experimental purposes), the run-time distribution information had to be read into SmartNet with the model-machine data. That meant altering the database file format to account for the run-time distribution data. Altering the database file format meant having to provide variables to hold the run-time distribution data, along with the functions necessary to retrieve and manipulate those variables. All of the run-time distribution variables and functions are first seen in `ModelMachine.h`. The changes made to this file are discussed below.

Because we referenced specific distribution function information, the distribution.h header file, written for this research and discussed later in this chapter, had to be included. We then added the class data members to hold the run-time distribution information. These data members were, of course, private. They include:

- `Distribution`: an Mstring type
- `Moment_1`: a float to hold the mean, or first moment
- `Moment_2`: a float to hold the second moment
- `Moment_3`: a float to hold the third moment

Public data member accessor functions were then declared. These functions include:

- `getDistribution()`: returns `Distribution`
- `getMoment_1()`: returns `Moment_1`
- `getMoment_2()`: returns `Moment_2`
- `getMoment_3()`: returns `Moment_3`

106

The above member function definitions were included as inline functions listed after the class definition. They are simple accessor functions that return the value of the individual data members.

A method had to be written that would allow a run-time duration to be generated based upon the new run-time distribution data members. By including it in the `ModelMachine` class, we had easy access to the necessary data. Also, when the actual duration is requested (see `server/simulator/JobStartEvent.cc` above) it is accessed via a reference to a `ModelMachine` type. We added the public member function `getRuntime()` to provide calculation of the run-time duration.

8. SERVER/SRC/MODELMACHINE.CC

This file contains member functions of the `ModelMachine` class. The class is defined in `ModelMachine.h`, discussed previously. Additions made to `ModelMachine.cc` include the following.

1. `ModelMachine::init()`: Added initialization of run-time distribution data members:

   - Distribution = " "
   - Moment_1 = 0.0;
   - Moment_2 = 0.0;
   - Moment_3 = 0.0;

2. `ModelMachine::operator=(ModelMachine &mm)`: Added assignment overloading for run-time distribution data members:

   - Distribution = mm.Distribution
   - Moment_1 = mm.Moment_1
   - Moment_2 = mm.Moment_2
   - Moment_3 = mm.Moment_3

3. `ModelMachine::getRuntime()`: This function was added to allow for the computation of the run-time duration. It returns duration, a DeltaTime type. The functions `generate_normal()` and `generate_exponential` were written for this research. They are defined in the file `distribution.h`, which is included in this file and discussed later in Appendix C. Here is the pseudo-code.

   - If Distribution is equal to "normal"
     - duration = generate_normal(Moment_1, Moment_2)
   - Else If Distribution is equal to "exponential"
     - duration = generate_exponential(Moment_1)
   - return duration

4. `ModelMachine::read()`: This function needed to be altered to allow for the run-time distribution information to be read in from the database file.

APPENDIX C. ADDITIONAL CODE FOR THE SMARTNET SIMULATOR

1. INTRODUCTION

The following subsections contain detailed information about the code we wrote specifically for improving the SmartNet simulator. Each explanation is followed by the actual code added to the SmartNet simulator.

2. SERVER/ARMSTRONG/MAKEFILE

The files that were written needed to be compiled with the SmartNet package. This meant creating a Makefile consistent with the Makefile structure resident in the SmartNet code. This file allows for all of the files below to be compiled whenever the server is recompiled. Here is the code: **Makefile**

```
# Makefile for Armstrong´s Thesis Code
# Used to generate Random Variates for
# use by the SmartNet simulator
# (last mod:  970518)
# Note that comments start with # for this file

# which compiler to use
CC = g++
#CC = CC

# Directory location of include files
#INCS = -I-L/local/lang/SC2.0.1
INCS =

CFLAGS = $(INCS) -g

# What libraries need to be linked
#LIBS = -lm
LIBS =

# Project name to be compiled
PROGS =
```

```
# What object files are to be used
OBJS  = distribution.o random_generator.o myrand.o

.SUFFIXES: .c .cc .o
.c.o:; cc $(CFLAGS) -c $*.c
.cc.o:; $(CC) $(CFLAGS) -c $*.cc

# What is to be compiled
#all : $(PROGS)
all : $(OBJS)
# The main object file
#mytest: $(OBJS)
#	$(CC) -o mytest $(OBJS) $(CFLAGS) $(LIBS)
# Note -- there is a tab before the $(CC) above

# What are the objects are dependent on
#main.o: main.cc proj2.h
#proj2.o: proj2.cc proj2.h
#main.o:  main.cc myrand.h random_generator.h distribution.h
distribution.o:  distribution.cc myrand.h random_generator.h
random_generator.o:  random_generator.cc random_generator.h
myrand.o:  myrand.cc myrand.h

# This cleans out everything except the Makefile,
# AAAREADME and source files
clean:; rm -f $(PROGS) *.o core
```

3. SERVER/ARMSTRONG/MYRAND.H & MYRAND.CC

The `myrand` files define a function that uniformly generates random numbers between 0 and 1. The uniform, randomly generated, number is used by later functions to access an array of 100 seeds that will assure high periodic randomization of numbers in another uniform random number generator. The pseudo-code of the `myrand()` function follows.

- static int check = false

- Use system time to seed the system random number generator

- If check is false

  - static tester = time(NULL)
  - check = true

- Seed the system random number generator with tester

- ix = system random number generator output

- answer = ix/(max random number capable of being generated)

- tester = ix

- return answer

The concept is for time to be used to first seed the random number generator. All subsequent calls to this function will use the previously generated variable as the seed because its location is kept intact via the static type. The reason the static typing was done is because there could possibly be several accesses of the `myrand()` function within a single second. Always using time for the seed would cause the same seed to be used for several `myrand()` calls. Here is the code: **myrand.h**

```
// File:  myrand.h
// Bob Armstrong
// 12 March 1997
// This function randomly generates numbers between
// 0 and 1
```

```cpp
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

typedef int bobint;

float myrand();
```

myrand.cc

```cpp
// File:  myrand.cc
// Bob Armstrong
// 12 March 1997
// This function uniformly generates random numbers between
// 0 and 1

#include "myrand.h"
#include <debug/Debug.h>

float myrand()
{
  long double ix;
  static long int tester;
  static bobint check = 0;

  // I am seeding the random function with the time
  srand((int)time(NULL));

  if (!check) {
    tester = time(NULL);
    //tester = 867875440;   // used to provide data consistency in testing
    check = 1;
    if(Debug::check("a1")){
      Debug::out() << "Initial seed (time):\t" << tester << endl;;
    }
  }

  srand((unsigned int)tester);
  ix = rand();  // make this the next time seed.
  float answer = ix/(RAND_MAX);
  tester = (long int)ix;
```

```
  if(Debug::check("a5")){
    Debug::out() << "Output of myrand:\t"<< answer << endl;
  }
  return answer;
}
```

```
  if(Debug::check("a5")){
    Debug::out() << "Output of myrand:\t"<< answer << endl;
  }
  return answer;
}
```

4. SERVER/ARMSTRONG/DISTRIBUTION.H & DISTRIBUTION.CC

The distribution files include most of the functions needed to generate the various run-time distributions. Functions include `normal_01()`, `generate_normal()`, and `generate_exponential()`.

The function `normal_01()` uses the polar method for generating `normal(0,1)` random variates. It has no parameters, but returns a single normally distributed random variate. This function is used by the `generate_normal()` function to generate Gaussian data based upon the first and second moments.

- While WW is greater than 1.0

  - $uniform\_random\_number\_1$ is a Uniform(0,1) random number
  - $uniform\_random\_number\_2$ is a Uniform(0,1) random number
  - $V1 = 2\ uniform\_random\_number\_1\ - 1$
  - $V2 = 2\ uniform\_random\_number\_2\ - 1$
  - $WW = V1^2 + V2^2$

- End While

- $YY = \sqrt{\frac{-2\log(WW)}{WW}}$

- $random\_variate\_1 = YY\ V1$

- $random\_variate\_2 = YY\ V2$

- Return either $random\_variate\_1$ or $random\_variate\_2$

The `generate_exponential()` function receives the first moment and returns a run-time duration. As explained in Chapter III, the Inverse Transform method is used to generate these exponentially distributed variates, because the exponential function, and its inverse, have a closed form.

- Define $EXPONENTIAL\_RUNTIME$

- While $duration$ is less than or equal to 0.0

  - $seed = 99\ myrand()$

114

- $random\_number = random\_generator(seed)$
- If first moment is greater than the $EXPONENTIAL\_RUNTIME$
  * $adjusted = first\_moment - EXPONENTIAL\_RUNTIME$
  * $duration = -EXPONENTIAL\_RUNTIME \ \log(random\_number)$
  * $duration = duration + adjusted$
- Else
  * $duration = -first\_moment \ \log(random\_number)$
- End If/Else

- End While

- Return $duration$

In this function, the constant $EXPONENTIAL\_RUNTIME$ is a mean gathered via experiments with the NAS Benchmarks which is applied to the first moment data specific to the machine/job pair. It is discussed in greater detail in Chapter V.

The `generate_normal()` function receives the first and second moment as its parameters and returns a run-time duration. This function calls the `normal_01()` function, which generated IID N(0,1) random variates. Implementation of the `normal_01()` function is simple, as shown in the following pseudo-code.

- $XX = normal\_01()$
- $duration = 0.5 + first\_moment + (XX \ \sqrt{second\_moment})$
- Return $duration$

The 0.5 is added to the duration computation to account for rounding errors. This function can be changed to generate truncated normal data by only allowing the duration to be returned if it falls within some limit imposed in the code. That limit may either be hard coded, or it may be dependent upon a constant relationship between the first and second moments, which is probably more realistic. The use of truncated Gaussian is discussed further in Chapter V. The code for these function is included below. **distribution.h**

`// File: distribution.h`

```
// Bob Armstrong
// 4 August 1997
// Thesis code
// This code determines which type of distribution
// the model-machine object carries with it and
// generates a run-time based upon that distribution.

#include <math.h>
#include <string.h>
#include "myrand.h"
#include "random_generator.h"
#include "/users/work3/rkarmstr/SOLARIS/src/sn/lib/spi/DeltaTime.h"

double normal_01();
DeltaTime generate_normal(float, float);
DeltaTime generate_exponential(float);
```

distribution.cc

```
// File:  distribution.cc
// Bob Armstrong
// 4 August 1997
// Thesis code
// This code determines which type of distribution
// the model-machine object carries with it and
// generates a run-time based upon that distribution.

#include "distribution.h"
#include <debug/Debug.h>


/* This is the polar method of generating normal
   random variates, discussed in Law and Kelton
   "Simulation Modeling and Analysis", pp 490 - 492.
 */
double normal_01()
{
  double random_variate;
  double v1, v2, yy, ww = 2.0;
  int seed;
  float random_number_1, random_number_2;
```

```cpp
  while(ww > 1.0) {
    seed = int(99 * myrand());

    if(Debug::check("a2")){
      Debug::out() << "Seed in normal_01():\t"<< seed << endl;
    }
    random_number_1 = random_generator(seed);
    random_number_2 = random_generator(seed);



    v1 = 2 * random_number_1 - 1;
    v2 = 2 * random_number_2 - 1;

    ww = v1 * v1 + v2 * v2;
  }

  yy = sqrt( (-2 * log(ww)) / ww);

  // Decide which value to return
  if(myrand() > 0.5) {
    random_variate = v1 * yy;
  }
  else {
    random_variate = v2 * yy;
  }

  if(Debug::check("a4")){
      Debug::out() << "Random Variate produced by normal_01():\t"
                   << random_variate << endl;
  }
  return random_variate;

}


DeltaTime generate_normal(float moment_1, float moment_2)
{

  DeltaTime duration;
  double xx;
  int checker = 0;
```

```
   double sigma = sqrt((double)moment_2);


   if(moment_2 == 0.0) {
      duration = moment_1;
   }
   else {
     while(checker == 0) {
       xx = normal_01();
       duration = (0.5 + moment_1 + sigma * xx);

       if((duration > 0.0) && (duration >= moment_1 - sigma)) {
checker = 1;
       }

   } // end while

  } // end else
  return duration;
}




DeltaTime generate_exponential(float moment_1)
{
  int seed;                        // holds seed for random_generator
  DeltaTime duration = -100;       // returned variable
  float adjusted;                  // moment_1 adjusted for EXP_RUNTIME
  float random_number;             // holds random_generator() value
  const float EXP_RUNTIME = 3.0;   // exponential mean; CHANGE THIS
                                   // to adjust exponential characteristics.


  // Only return a runtime duration > 0.
  // Everything takes SOME time to run!
  while(duration <= 0){
    // Get seed and generate random number
    seed = int(99 * myrand());
    random_number = random_generator(seed);
```

```
    // If moment_1 is greater than the runtime value,
    // subtract moment_1 and compute the duration from
    if(moment_1 > EXP_RUNTIME) {
      adjusted = moment_1 - EXP_RUNTIME;
      duration = (int)(- EXP_RUNTIME * log(random_number));
      duration += (DeltaTime)adjusted;
    } else {
      duration = (int)(- moment_1 * log(random_number));
    }
  } // end while

  return duration;
}
```

5. SERVER/ARMSTRONG/RANDOM_GENERATOR.H & RANDOM_GENERATOR.CC

This file contains the functions necessary to generate uniformly distributed IID U(0,1) random variates. This function is needed by the `normal_01()`, `generate_normal()`, and `generate_exponential()` functions found in the distribution files. As has been previously discussed, a good source of IID U(0,1) random variables is essential to the success of any random generator. The following code can be found written in "Simulation Modeling and Analysis," by Law and Kelton. [Ref. 13, pages 454-456] It is also included below. **random-generator.h**

```
/* The following 3 declarations are for use of the random-number
   generator rand and the associated functions randst and reandgt for
   seed management.  This file (named random_generator.c) should be
   included in any program using these functions by executing
   #include "random_generator.h"
   before referencing the functions.
   */

float random_generator(int stream);
void randst(long zset, int stream);
long randgt(int stream);
```

random-generator.cc

```
/* File: random_generator.cc
   UNIFORM (0,1) RANDOM NUMBER GENERATOR
   Stolen by: Bob Armstrong from "Simulation
   Modeling and Analysis", by Law and Kelton */

/* Prime modulus multiplicative linear congruential generator Z[i] =
   (630360016 * Z[i-1]) (mod{pow(2, 31) - 1)), based upon Marse and
   Roberts´ portable FORTRAN random-number generator UNIRAN.  Multiple
   (100) streams are supported, with seess spaced 100,000 apart.
   Throughout, input argument "stream" must be an int giving the
   desired stream input number.  The header file random_generator.h
   must be included in the calling program (#include
   "random_generator.h") before using these functions.
```

```
        Usage:  (three functions)


        1.  To obtain the next U(0,1) random number from stream "stream,"
        execute u = random_generator(stream);
        where rand is a float function.  The float variable u will contain the next
        random number.


        2.  To set the seed for stream "stream," to a desired value zset,
        execute randst(zset, stream);
        where randst is a void function and zset must be a long set to
        the desired seed, a number between 1 and 2147483646 (inclusive).
        Default seeds for all 100 streams are given in the code.


        3.  To get the current (most recently used) integer in the sequence being
        generated for stream "stream" into the long variable zget,
        execute zget = randgt(stream);
        where randgt is a long function.  */

#include <iostream.h>
#include <debug/Debug.h>
/* Define the constants. */

#define MODLUS 2147483647
#define MULT1              24112
#define MULT2              26143


/* Set the default seeds for all 100 streams. */

static long zrng[] =
{ 0,
   1973272912, 281629770,  20006270,   1280689831, 2096730329, 1933576050,
   913566091,  246780520,  1363774876, 604901985,  1511192140, 1259851944,
   824064364,  150493284,  242708531,  75253171,   1964472944, 1202299975,
   233217322,  1911216000, 726370533,  403498145,  993232223,  1103205531,
   762430696,  1922803170, 1385516923, 76271663,   413682397,  726466604,
   336157058,  1432650381, 1120463904, 595778810,  877722890,  1046574445,
   68911991,   2088367019, 748545416,  622401368,  2122378830, 640690903,
   1774806513, 2132545692, 2079249579, 78130110,   852776735,  1187867272,
   1351423507, 1645973084, 1997049139, 922510944,  2045512870, 898585771,
   243649545,  1004818771, 773686062,  403188473,  372279877,  1901633463,
   498067494,  2087759558, 493157915,  597104727,  1530940798, 1814496276,
   536444882,  1663153658, 855503735,  67784357,   1432404475, 619691088,
```

```
        119025595,  880802310,  176192644,  1116780070, 277854671,  1366580350,
        1142483975, 2026948561, 1053920743, 786262391,  1792203830, 1494667770,
        1923011392, 1433700034, 1244184613, 1147297105, 539712780,  1545929719,
        190641742,  1645390429, 264907697,  62038953,   1502074852, 927711160,
        364849192,  2049576050, 638580085,  547070247 };

/* Generate the next random number. */

float random_generator(int stream)
{
  long zi, lowprd, hi31;

    if(Debug::check("a6")){
      Debug::out() << "Seed into random_generator:\t" << zrng[stream] << endl;
  }

  zi = zrng[stream];
  lowprd = (zi & 65535) * MULT1;
  hi31 = (zi >> 16) * MULT1 + (lowprd >> 16);
  zi = ((lowprd & 65535) - MODLUS) + ((hi31 & 32767) << 16) + (hi31 >> 15);
  if(zi < 0) {
    zi += MODLUS;
  }
  lowprd = (zi &65535) * MULT2;
  hi31 = (zi >> 16) * MULT2 + (lowprd >> 16);
  zi = ((lowprd & 65535) - MODLUS) + ((hi31 & 32767) << 16) + (hi31 >> 15);
  if(zi < 0) {
    zi += MODLUS;
  }
  zrng[stream] = zi;
  if(Debug::check("a3")){
      Debug::out() << "Output from random_generator:\t"
                   << ((zi >> 7 | 1) + 1)/16777216.0 << endl;
  }
  return ((zi >> 7 | 1) + 1)/16777216.0;
}




/* Set the current zrng for stream "stream" to zset. */
void randst (long zset, int stream)
{
```

```
  zrng[stream] = zset;
}



/* Return the current zrng for stream "stream" */

long randgt(int stream)
{
  return zrng[stream];
}
```

APPENDIX D. CODE FOR RUNTIME DISTRIBUTION TESTS

1.   CODE FOR COUNTING SORT

The following code is my implementation of the NAS Integer Sort Benchmark. It is written to be run on an SGI machine with four processors. The sorting algorithm used is a parallel version of the counting sort. The code also includes a non-parallel version of counting sort, which was run to provide a comparison for speedup.

```
/****************************************************************
File:  parallel4.c
Name:  Bob Armstrong
Purpose:  This file contains functions executed in the main
          procedure that measurement of the counting sort
          executed in sequence on one processor, in sequence
          forked to one processor, and in parallel forked
          to four processors.  The code is written for the
          SGI Challenge L.  Measurements are taken and output
          to three files (one for each treatment) for each of
          ten runs of the sort.

          The code is not to the NPS style guide (sue me).

****************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <ulocks.h>
#include <unistd.h>
#include <stddef.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/syssgi.h>

#define   TOTAL_KEYS_LOG_2     22
#define   MAX_KEY_LOG_2        11
#define   TOTAL_KEYS           (1 << TOTAL_KEYS_LOG_2)
#define   MAX_KEY              (1 << MAX_KEY_LOG_2)
```

```c
#define CYCLE_COUNTER_IS_64BIT 1

#if CYCLE_COUNTER_IS_64BIT
  typedef unsigned long long iotimer_t;
#else
  typedef unsigned int iotimer_t;
#endif

/* These are globals to make the arrays, which are accessed
   randomly, available to all functions.  This decreases
   the time spent passing pointers.
 */
int key_array[TOTAL_KEYS];
int work_array[MAX_KEY];
int final_array[TOTAL_KEYS];

/* This is the LOCK stuff. */
usptr_t* handle = NULL;
ulock_t lock_array[MAX_KEY];

/* These are globals to hold the values in work_array
   after the tallys are done in parallel.  They
   need to be globals because I can only pass 6 parameters
   in the m_fork call.
 */
int data1 = 0;
int data2 = 0;
int data3 = 0;


/*  This is Pedro Tsai's way cool precision timer for SGI machines.
    It was originally written in C++.  With MINOR changes, it is
    included here to compile as C code.  The units returned by the
    gethrtimer() function are picoseconds.  Thanks, Pedro!
 */
unsigned int cycleval;

volatile iotimer_t *iotimer_addr;
static int initflag=0;

volatile iotimer_t* initSysTimer()
{
```

```
    __psunsigned_t phys_addr, raddr;
    int fd, poffmask;

    if ( initflag==0 )
    {
      poffmask = getpagesize() - 1;

      phys_addr = syssgi(SGI_QUERY_CYCLECNTR, &cycleval);


      raddr = phys_addr & ~poffmask;
      fd = open("/dev/mmem", O_RDONLY);
      iotimer_addr = (volatile iotimer_t *)mmap(0, poffmask, PROT_READ,
                  MAP_PRIVATE, fd, (off_t)raddr);
      iotimer_addr = (iotimer_t *)((__psunsigned_t)iotimer_addr +
                  (phys_addr & poffmask));
      initflag=1;
    }
    return iotimer_addr;
}

/* get the hardware counter value */
long long gethrtime()
{
    volatile iotimer_t *timer_addr;
    long long counter_value;

    /* Initialize the hardware time counter */
    timer_addr=initSysTimer();

    counter_value=*timer_addr;
    return counter_value;

}


/*
 *    FUNCTION RANDLC (X, A)
 *
 * This routine returns a uniform pseudorandom double precision number in the
 * range (0, 1) by using the linear congruential generator
 *
```

```
*   x_{k+1} = a x_k   (mod 2^46)
*
*   where 0 < x_k < 2^46 and 0 < a < 2^46.  This scheme generates 2^44 numbers
*   before repeating.  The argument A is the same as 'a' in the above formula,
*   and X is the same as x_0.  A and X must be odd double precision integers
*   in the range (1, 2^46).  The returned value RANDLC is normalized to be
*   between 0 and 1, i.e. RANDLC = 2^(-46) * x_1.  X is updated to contain
*   the new seed x_1, so that subsequent calls to RANDLC using the same
*   arguments will generate a continuous sequence.
*
*   This routine should produce the same results on any computer with at least
*   48 mantissa bits in double precision floating point data.  On Cray systems,
*   double precision should be disabled.
*
*   David H. Bailey    October 26, 1990
*
*       IMPLICIT DOUBLE PRECISION (A-H, O-Z)
*       SAVE KS, R23, R46, T23, T46
*       DATA KS/0/
*
*   If this is the first call to RANDLC, compute R23 = 2 ^ -23, R46 = 2 ^ -46,
*   T23 = 2 ^ 23, and T46 = 2 ^ 46.  These are computed in loops, rather than
*   by merely using the ** operator, in order to insure that the results are
*   exact on all systems.  This code assumes that 0.5D0 is represented exactly.
*/


/***************************************************************/
/************           R  A  N  D  L  C          ***********/
/************                                     ***********/
/************    portable random number generator   ***********/
/***************************************************************/

double randlc(X, A)
double *X;
double *A;
{
        static int        KS;
        static double R23, R46, T23, T46;
        double T1, T2, T3, T4;
        double A1;
        double A2;
```

```
      double X1;
      double X2;
      double Z;
      int     i, j;

      if (KS == 0)
      {
        R23 = 1.0;
        R46 = 1.0;
        T23 = 1.0;
        T46 = 1.0;

        for (i=1; i<=23; i++)
        {
          R23 = 0.50 * R23;
          T23 = 2.0 * T23;
        }
        for (i=1; i<=46; i++)
        {
          R46 = 0.50 * R46;
          T46 = 2.0 * T46;
        }
        KS = 1;
      }

/*  Break A into two parts such that A = 2^23 * A1 + A2 and set X = N.  */

      T1 = R23 * *A;
      j  = T1;
      A1 = j;
      A2 = *A - T23 * A1;

/*  Break X into two parts such that X = 2^23 * X1 + X2, compute
    Z = A1 * X2 + A2 * X1   (mod 2^23), and then
    X = 2^23 * Z + A2 * X2   (mod 2^46).                        */

      T1 = R23 * *X;
      j  = T1;
      X1 = j;
      X2 = *X - T23 * X1;
      T1 = A1 * X2 + A2 * X1;
```

```
        j   = R23 * T1;
        T2 = j;
        Z = T1 - T23 * T2;
        T3 = T23 * Z + A2 * X2;
        j   = R46 * T3;
        T4 = j;
        *X = T3 - T46 * T4;
        return(R46 * *X);
}
/* end randlc(x,a) */



/****************************************************************/
/************* C  R  E  A  T  E  _  S  E  Q     ***********/
/****************************************************************/

/* This function creates the sequence of keys that will be sorted
   by calling the random number generator previously explained
   in this file.  It is stored in key_array.
 */
void create_seq( double seed, double a )
{
double x;
int    i, j, k;

        k = MAX_KEY/4;

for (i=0; i<TOTAL_KEYS; i++)
{
    x = randlc(&seed, &a);
    x += randlc(&seed, &a);
    x += randlc(&seed, &a);
    x += randlc(&seed, &a);

            key_array[i] = k*x;
        }
}


/****************************************************************/
/****************** COUNTING_SORT ***************************/
/****************************************************************/
/* This function is used to fill the final_array with the sorted keys.
```

```
      It is not the entire counting sort algorithm.
  */
void counting_sort(begin, end, div1, div2, div3)
int begin;
int end;
int div1;
int div2;
int div3;
{
  int ix, aa;

  for(ix = end-1 ; ix > begin - 1; ix--) {
    aa = key_array[ix];
    final_array[work_array[aa]-1] = aa;
    work_array[aa]--;
  }
  return;
}



/*******************************************************************/
/************************* DO_SORT *******************************/
/*******************************************************************/
/* This function, like counting_sort above, only fills the final_array
   with the sorted keys.  It is a function meant to be called with
   the m_fork() function call specific to SGI machines.
  */
void do_sort(div1, div2, div3, dd1, dd2, dd3)
int div1;
int div2;
int div3;
int dd1;
int dd2;
int dd3;
{
  int ix, iy, iw, iz, aa, ab, ac, ad,
      wa, wb, wc, wd;
  ulock_t* ba,* bb,* bc,* bd;

  if(m_get_myid() == 0) {
    for(ix = div1-1 ; ix > -1; ix--) {
      aa = key_array[ix];
```

```
      while(ustestlock(lock_array[aa])) {

      }
      ussetlock(lock_array[aa]);
      wa = work_array[aa];
      if(aa < dd1) {
final_array[wa-1] = aa;
      }else if(aa < dd2) {
final_array[wa + data1 - 1] = aa;
      }else if(aa < dd3) {
final_array[wa + data2 + data1 - 1] = aa;
      }else {
final_array[wa + data3 + data2 + data1 - 1] = aa;
      }
      work_array[aa]--;
      usunsetlock(lock_array[aa]);
    }
  }else if(m_get_myid() == 1) {
    for(iy = div2-1 ; iy > div1-1; iy--) {
      ab = key_array[iy];
      while(ustestlock(lock_array[ab])) {

      }
      ussetlock(lock_array[ab]);
      wb = work_array[ab];
      if(ab<dd1) {
final_array[wb-1] = ab;
      }else if(ab<dd2) {
final_array[wb + data1 - 1] = ab;
      }else if(ab<dd3) {
final_array[wb + data2 + data1 - 1] = ab;
      }else {
final_array[wb + data3 + data2 + data1 - 1] = ab;
      }
      work_array[ab]--;
      usunsetlock(lock_array[ab]);
    }
  }else if(m_get_myid() == 2) {

    for(iz = div3-1 ; iz > div2-1; iz--) {
      ac = key_array[iz];
      while(ustestlock(lock_array[ac])) {
```

```
        }
        ussetlock(lock_array[ac]);
        wc = work_array[ac];
        if(ac<dd1) {
final_array[wc-1] = ac;
        }else if(ac<dd2) {
final_array[wc + data1 - 1] = ac;
        }else if(ac<dd3) {
final_array[wc + data2 + data1 - 1] = ac;
        }else {
final_array[wc + data3 + data2 + data1 - 1] = ac;
        }
        work_array[ac]--;
        usunsetlock(lock_array[ac]);
      }
    }else if(m_get_myid() == 3) {
      for(iw = TOTAL_KEYS-1 ; iw > div3-1; iw--) {
        ad = key_array[iw];
        while(ustestlock(lock_array[ad])) {

        }
        ussetlock(lock_array[ad]);
        wd = work_array[ad];

        if(ad<dd1) {
final_array[wd-1] = ad;
        }else if(ad<dd2) {
final_array[wd + data1 - 1] = ad;
        }else if(ad<dd3) {
final_array[wd + data2 + data1 - 1] = ad;
        }else {
final_array[wd + data3 + data2 + data1 - 1] = ad;
        }
        work_array[ad]--;
        usunsetlock(lock_array[ad]);
      }
    }
    return;
}
/****************************************************************/
/*********************** set_zero ***************************/
```

```
/****************************************************************/
/*  This function sets every element of the work_array to zero.
    This function is meant to be called by the m_fork function
    specific to SGI machines.
 */
void set_zero(div1, div2, div3)
int div1;
int div2;
int div3;
{
  int ix, iy, iz, iw;

  if(m_get_myid() == 0) {
    for(ix = 0; ix < div1; ix++)
      work_array[ix] = 0;
  }else if(m_get_myid() == 1) {
    for(iy = div1; iy < div2; iy++)
      work_array[iy] = 0;
  }else if(m_get_myid() == 2) {
    for(iz = div2; iz < div3; iz++)
      work_array[iz] = 0;
  }else if(m_get_myid() == 3) {
    for(iw = div3; iw < MAX_KEY; iw++)
      work_array[iw] = 0;
  }
  return;
}


/****************************************************************/
/*********************** verify ** ***************************/
/****************************************************************/
int verify()
{
  int ix, check;

  for(ix = 1; ix < TOTAL_KEYS; ix++) {
    if(final_array[ix] < final_array[ix-1]) {
      check = 1;
      break;
    }
    else { check = 0;}
  }
```

```
  return check;
}


/****************************************************************/
/************************* increment ****************************/
/****************************************************************/
/* This function counts the occurances of a KEY by incrementing the
   value of work_array[KEY].  Thusly, work_array[4] will contain a count
   of the number of keys that are the number 4.  This function is
   meant to be called using the SGI function m_fork.  It is set up
   for parallel execution.
 */
void increment(div1, div2, div3)
int div1;
int div2;
int div3;
{
   int ix, iy, iz, iw, aa, ab, ac, ad;
   /*ulock_t    ba, bb, bc, bd;*/

   if(m_get_myid() == 0) {
     for(ix = 0; ix < div1; ix++){
       aa = key_array[ix];
       while(ustestlock(lock_array[aa])) {

       }
       ussetlock(lock_array[aa]);
       work_array[aa]++;
       usunsetlock(lock_array[aa]);
     }
   }else if(m_get_myid() == 1) {
     for(iy = div1; iy < div2; iy++){
       ab = key_array[iy];
       while(ustestlock(lock_array[ab])) {

       }
       ussetlock(lock_array[ab]);
       work_array[ab]++;
       usunsetlock(lock_array[ab]);
     }
   }else if(m_get_myid() == 3) {
```

```
      for(iz = div2; iz < div3; iz++){
        ac = key_array[iz];
        while(ustestlock(lock_array[ac])) {

        }
        ussetlock(lock_array[ac]);
        work_array[ac]++;
        usunsetlock(lock_array[ac]);
      }
    }else {
      for(iw = div3; iw < TOTAL_KEYS; iw++){
        ad = key_array[iw];
        while(ustestlock(lock_array[ad])) {

        }
        ussetlock(lock_array[ad]);
        work_array[ad]++;
        usunsetlock(lock_array[ad]);
      }
    }
    return;
}



/****************************************************************/
/************************ tally *******************************/
/****************************************************************/
/*  This function tallys the number of work_array elements less than
    or equal to the work_array index.  This function is called
    by the SGI function m_fork for 4 processors.
 */
void tally(div1, div2, div3)
int div1;
int div2;
int div3;
{
  int ix, iy, iz, iw;

  if(m_get_myid() == 0) {
    for(ix = 1; ix < div1; ix++)
      work_array[ix] += work_array[ix - 1];
  }else if(m_get_myid() == 1) {
```

```
      for(iy = div1+1; iy < div2; iy++)
        work_array[iy] += work_array[iy - 1];
    }else if(m_get_myid() == 2) {
      for(iz = div2+1; iz < div3; iz++)
        work_array[iz] += work_array[iz - 1];
    }else if(m_get_myid() == 3) {
      for(iw = div3+1; iw < MAX_KEY; iw++)
        work_array[iw] += work_array[iw - 1];
    }
    return;
}


/******************************************************************/
/******************** MAIN PROGRAM *******************************/
/******************************************************************/
main()
{
    double      xx, aa, zz;
    long long   duration, end, stop, one, two, three, four, szo, inc, srt, tal;
    FILE        *true_sequential, *fork_sequential, *forked;
    float       data;
    int         ix, iy, yy, dd, dd1, dd2, dd3,
                division, div1, div2, div3;
    char*       lock_file = "lock_file";
    unsigned int MAX = 400000;

    /* set up output files */
    true_sequential = fopen("tsequential.dat", "w");
    forked          = fopen("forked.dat", "w");

    /* Create a sequence of keys to sort */
    create_seq( 314159265.00, 1220703125.00 );

    /* calculate array boundaries for key_array final_array */
    division = TOTAL_KEYS/4;
    div1    = division;
    div2    = div1 + division;
    div3    = div2 + division;

    /* calculate array boundaries for work_array */
    dd  = MAX_KEY/4;
```

```
dd1 = dd;
dd2 = dd1 + dd;
dd3 = dd2 + dd;

/* Set up lock configuration and handle information */
usconfig(CONF_INITSIZE, MAX);
handle = usinit(lock_file);

/* Initialize the lock_array (first time)*/
for(ix = 0; ix < MAX_KEY; ix++) {
  lock_array[ix] = usnewlock(handle);
  usinitlock(lock_array[ix]);
}

/* Initialize memory by running the sequential sort once */
m_fork(set_zero, dd1, dd2, dd3);
for(ix = 0; ix < TOTAL_KEYS; ix++) {
  work_array[key_array[ix]]++;
}
for(ix = 1; ix < MAX_KEY; ix++) {
  work_array[ix] += work_array[ix - 1];
}
counting_sort(0, TOTAL_KEYS);




/*  Run the sort sequentially (single processor)
    as a baseline measurement for speedup.
 */

for(iy = 0; iy < 1000; iy++) {
  end = gethrtime();                      /* start time */

  /* initialize work_array to zero */
  for(ix = 0; ix < MAX_KEY; ix++) {
    work_array[ix] = 0;
  }

  /* count occurances of each key being sorted */
  for(ix = 0; ix < TOTAL_KEYS; ix++) {
    work_array[key_array[ix]]++;
```

```
    }

    /* count the elements in work_array less than or equal to ix */
    for(ix = 1; ix < MAX_KEY; ix++) {
      work_array[ix] += work_array[ix - 1];
    }

    /* sort the keys into final_array */
    counting_sort(0, TOTAL_KEYS);

    stop = gethrtime();

    /* Verify proper sorting */
    if(verify()) {
      printf("True Sequential Final-Array (run %d) failed verification!\n", iy);
    }
    else {
      printf("True Sequential Final-Array (run %d) passed verification!\n", iy);
    }

    duration = stop - end;                   /* calculate duration */
    data = (float)duration/1000000000;    /* convert duration to seconds */
    fprintf(true_sequential, "Optimum Sequential sort time is:  %f\n", data);
} /* end for */

fclose(true_sequential);

/* set number of processors to 4 */
m_set_procs(4);
/* Initialize memory by running the forked sort once */
m_fork(set_zero, dd1, dd2, dd3);
m_fork(increment, div1, div2, div3);
m_fork(tally, dd1, dd2, dd3);
m_fork(do_sort, div1, div2, div3, dd1, dd2, dd3);

/* Perform the counting sort using forking
   and all 4 processors.  This is what we
   "hope" provides speedup.
 */
for(iy = 0; iy < 1000; iy++) {
  end = gethrtime();                         /* start time */
```

```
/* initialize work_array to zero */
m_fork(set_zero, dd1, dd2, dd3);
one = gethrtime();
/* count occcurances of each key being sorted */
m_fork(increment, div1, div2, div3);
two = gethrtime();
/* count the elements in work_array less than or equal to ix */
m_fork(tally, dd1, dd2, dd3);
three = gethrtime();
/* Record tally sums (at the upper interval limit) in globals */
data1 = work_array[dd1 - 1];
data2 = work_array[dd2 - 1];
data3 = work_array[dd3 - 1];
four = gethrtime();
/* sort the keys into final_array */
m_fork(do_sort, div1, div2, div3, dd1, dd2, dd3);

stop = gethrtime();

if(verify()) {
  printf("Fully Forked Final-Array (run %d) failed verification!\n", iy);
  exit(1);
}
else {
  printf("Fully Forked Final-Array (run %d) passed verification!\n", iy);
}

duration = stop - end;          /* calculate duration */
szo = one - end;
inc = two - one;
tal = three - two;
srt = stop - four;
data = (float)duration/1000000000;      /* convert duration to seconds*/
fprintf(forked, "Forked sort time is:  %f\n", data);
data = (float)szo/1000000000;
fprintf(forked, "Time spent in set_zero: \t %f \n", data);
data = (float)inc/1000000000;
fprintf(forked, "Time spent in increment:\t %f \n", data);
data = (float)tal/1000000000;
fprintf(forked, "Time spent in tally:    \t %f \n", data);
data = (float)srt/1000000000;
fprintf(forked, "Time spent in do_sort:  \t %f \n", data);
```

```
    }

    fclose(forked);

    return 0;
}
```

APPENDIX E. SIMULATION EXPERIMENTAL DATA

1. HETEROGENEITY QUADRANT DATA

The tables included in this appendix are the "shorthand" matrices refered to in Chapter V.

		Machine				
Job		1	2	3	4	5
1	mean	30034	11	239	30097	533
2	mean	25	1003	8619	75	65037
3	mean	1078	93	1950	204001	8081
4	mean	35096	9501	29	2582	1000
5	mean	63	45055	1074075	11533	15
		Machine				
Job		6	7	8	9	10
1	mean	69	42799	1396	52453	4652
2	mean	30093	4723	11372	16333	287
3	mean	233	9	193	566	63526
4	mean	75019	23333	782	1134	1705
5	mean	403	207	6374	304291	666

Table XII. High-Job, High-Machine Heterogeneity.

		Machine				
Job		1	2	3	4	5
1	mean	25	26	27	28	29
2	mean	175	166	174	167	173
3	mean	3095	3094	3009	3096	3093
4	mean	9900	9899	9898	9897	9896
5	mean	30007	30006	30005	30004	30003
		Machine				
Job		6	7	8	9	10
1	mean	30	31	32	33	34
2	mean	168	172	169	171	170
3	mean	3097	3092	3098	3091	3099
4	mean	9901	9902	9903	9904	9905
5	mean	30002	30001	30000	30008	30009

Table XIII. High-Job, Low-Machine Heterogeneity.

		Machine				
Job		1	2	3	4	5
1	mean	5	1003	101	29	2002
2	mean	6	1001	104	25	2001
3	mean	9	1002	102	27	2000
4	mean	8	1000	103	28	2004
5	mean	7	1004	100	26	2003
		Machine				
Job		6	7	8	9	10
1	mean	69	5500	300	9996	25
2	mean	65	5499	299	10000	22
3	mean	67	5497	298	9998	23
4	mean	66	5498	297	9999	21
5	mean	68	5496	296	9997	24

Table XIV. Low-Job, High-Machine Heterogeneity.

	Machine				
Job	1	2	3	4	5
1 mean	23	22	21	20	24
2 mean	24	23	22	21	25
3 mean	25	24	23	22	26
4 mean	26	25	24	23	27
5 mean	27	26	25	24	28

	Machine				
Job	6	7	8	9	10
1 mean	25	27	28	31	30
2 mean	26	25	26	28	20
3 mean	27	23	24	25	28
4 mean	28	21	22	22	22
5 mean	29	19	20	19	25

Table XV. Low-Job, Low-Machine Heterogeneity.

	Machine				
Job	1	2	3	4	5
1 mean	300034	52453	42799	30097	4652
2 mean	65037	30093	16333	11372	8619
3 mean	204001	63526	8081	1950	1078
4 mean	75019	35096	23333	9501	2582
5 mean	1074075	304291	11533	6374	666

	Machine				
Job	6	7	8	9	10
1 mean	1396	533	239	69	11
2 mean	4723	1003	287	75	25
3 mean	566	233	193	93	9
4 mean	1705	1134	1000	782	29
5 mean	6374	403	207	63	15

Table XVI. High-Job, High-Machine, Consistent Heterogeneity.

		Machine				
Job		1	2	3	4	5
1	mean	9996	5500	2002	1003	300
2	mean	10000	5499	2001	1001	299
3	mean	9998	5497	2000	1002	298
4	mean	9999	5498	2004	1000	297
5	mean	9997	5496	2003	1004	296
		Machine				
Job		6	7	8	9	10
1	mean	101	69	29	25	5
2	mean	104	65	25	22	6
3	mean	102	67	27	23	9
4	mean	103	66	28	21	8
5	mean	100	68	26	24	7

Table XVII. Low-Job, High-Machine, Consistent Heterogeneity.

APPENDIX F. SIMULATION EXPERIMENT RESULTS

1.   ZERO-VARIANCE SIMULATION EXPERIMENT RESULTS

125-1	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	1,074,104	119,576	10,000	314	204,001	9,998
LBA	783	690,000	883	1,094	2,221	883
Greedy	782	105,620	483	289	1,666	483
Fastgreedy	783	119,454	507	293	1,664	505
125-2	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	1,074,074	122,491	9,996	319	75,019	10,000
LBA	754	750,000	869	964	2,291	869
Greedy	754	109,500	472	288	1,681	472
Fastgreedy	754	122,396	491	299	1,669	494
500-3	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	1,074,075	458,809	9,996	1,230	204,001	9,997
LBA	2,842	3,090,000	3,497	4,240	8,834	3,497
Greedy	2,726	439,018	1,874	1,119	6,514	1,874
Fastgreedy	2,697	458,910	1,931	1,158	6,498	1,931
500-4	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	1,082,723	430,923	9,998	1,235	304,291	9,996
LBA	2,813	2,880,000	3,485	4,300	8,858	3,485
Greedy	2,697	416,990	1,865	1,116	6,527	1,865
Fastgreedy	2,639	430,781	1,930	1,155	6,506	1,924

Table XVIII. Baseline Simulation Experiment Results. Heterogeneity should be read Job-Machine. Also, "Con" refers to consistency; absence of "Con" means the heterogeneity is inconsistent.

2. RESULTS OF SIMULATION EXPERIMENTS WHERE JOBS RAN FOR TIMES DIFFERENT FROM PREDICTED TIMES.

a. Exponential Run-time Distribution Experiment Results

125-1	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	9,999.53	203,999.80	9,994.13
LBA	873.27	2,189.73	851.00
Greedy	497.93	1,657.53	489.33
Fastgreedy	523.00	1,643.87	538.93
125-2	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	9,994.80	75,020.47	9,995.93
LBA	854.87	2,289.00	846.40
Greedy	481.80	1,662.13	492.33
Fastgreedy	705.13	1,642.20	502.07
500-3	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	9,995.13	204,001.27	9,995.93
LBA	3,470.33	8,805.07	3,482.00
Greedy	1,905.13	6,504.20	1,899.53
Fastgreedy	1,956.53	6,507.93	1,967.33
500-4	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	9,998.20	304,291.67	9,995.60
LBA	3,458.20	8,854.07	3,459.07
Greedy	1,905.60	6,565.33	1,927.13
Fastgreedy	1,964.80	6,521.60	1,956.33

Table XIX. Exponential Experiment Results for the Low-Job, High-Machine, High-Job, High-Machine, Consistent, and Low-Job, High-Machine, Consistent categories of heterogeneity.

b. Truncated Gaussian Run-time Distribution Experiment Results

125-1	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	10,032.93	300,807.00	10,066.73
LBA	1,054.13	2,477.47	1,055.40
Greedy	594.93	1,871.87	572.87
Fastgreedy	603.80	1,839.00	594.93
125-2	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	10,029.27	300,053.20	10,045.60
LBA	1,032.07	2,530.53	1,037.87
Greedy	574.20	1,879.87	564.27
Fastgreedy	593.13	1,885.40	603.53
500-3	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	10,092.27	304,570.27	10,045.40
LBA	4,251.40	9,983.20	4,247.00
Greedy	2,298.40	7,288.93	2,295.00
Fastgreedy	2,343.80	7,269.93	2,341.33
500-4	Lo-Hi	Hi-Hi-Con	Lo-Hi-Con
OLB	10,056.73	1,074,996.07	10,032.47
LBA	4,250.27	9,988.60	4,209.93
Greedy	2,285.47	7,342.80	2,275.73
Fastgreedy	2,357.47	7,304.87	2,336.07

Table XX. Truncated Gaussian Experiment Results for the Low-Job, High-Machine, High-Job, High-Machine, Consistent, and Low-Job, High-Machine, Consistent categories of heterogeneity.

3.   ADDITIONAL EXPERIMENTS
a.   Comparison of Baseline Run-time and Theoretical Best Case Run-time

125-1	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	483512%	14%	11225%	22%	91750%	11222%
LBA	252%	561%	900%	327%	900%	900%
Greedy	252%	1%	7 447%	12%	650%	447%
Fastgreedy	252%	14%	474%	14%	649%	471%
125-2	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	468723%	13%	11402%	25%	32645%	11407%
LBA	229%	595%	900%	278%	900%	900%
Greedy	229%	1%	443%	13%	633%	443%
Fastgreedy	229%	13%	465%	17%	628%	468%
500-3	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	121484%	4%	2758%	20%	22992%	2758%
LBA	221%	605%	900%	315.89%	900%	900%
Greedy	208%	0.24%	435%	9%	637%	435%
Fastgreedy	205%	4%	452%	13%	635%	452%
500-4	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
OLB	122131%	3%	2768%	21%	34252%	2768%
LBA	217%	592%	900%	321%	900%	900%
Greedy	204%	0.23%	435%	9%	636%	435%
Fastgreedy	197%	3%	453%	13%	634%	452%

Table XXI. Theoretical Best versus Baseline Completion Time.. This data depicts the percentage difference between the theoretical *Best Case Time* and the baseline completion time. In every case, SmartNet builds a schedule which takes longer to execute than the theoretical *Best Case Time*.

b.   Greedy versus Fast Greedy Performance

Test	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
Baseline	-0.77%	8.18%	3.88%	3.05%	-0.35%	3.86%
Exponential			14.30%		-0.66%	4.30%
T-Gaussian			2.48%		-0.56%	3.87%

Table XXII. Greedy versus Fast Greedy, Sequential Method 145 . This table shows how much faster schedules built by the Greedy algorithm finish executing versus schedules built by the Fast Greedy algorithm using the Sequential Method of job request. Positive values mean that the Greedy schedule is executed xx% faster than the Fast Greedy schedule.

c.   Grouped versus Sequential Job Request Methods

	Hi-Hi	Hi-Lo	Lo-Hi	Lo-Lo	Hi-Hi-Con	Lo-Hi-Con
Grouped Method	19.95%	2.44%	5.37%	8.12%	4.58%	5.37%

Table XXIII. Greedy versus Fast Greedy, Grouped Method. This table shows how much faster schedules built by the Greedy algorithm finish executing versus schedules built by the Fast Greedy algorithm. Positive values mean that the Greedy schedule is executed xx% faster than the Fast Greedy schedule.

APPENDIX G. HOW TO RUN SMARTNET

1. GETTING STARTED
a. Unpacking the Code

It is suggested by the SmartNet development team that the code be unpacked into a directory called SOLARIS. We follow that advice throughout this appendix. The name SOLARIS is used because we used the Solaris operating system version of SmartNet, and hence compiled the code on a Solaris machine. Take the sn.tar.gz file, move it into the SOLARIS directory, and unzip it. Next, execute the command

```
tar xvf sn.tar
```

and the source code will expand.

b. Setting the Environment

In order to compile and run SmartNet, your environment must be set properly. Below is all that I needed to do to set my environment for use at NPS (my login name was `rkarmstr`; substitute your path and login name as appropriate).

```
# setup for SmartNet setenv SNROOT
# /users/work3/rkarmstr/SOLARIS set path=($path
# /users/work3/rkarmstr/SOLARIS/local/bin) set path=($path
# /opt/cygnus/bin) set path=($path /usr/xpg4/bin) setenv
# LD_LIBRARY_PATH /usr/include\:$LD_LIBRARY_PATH
```

c. Compiling SmartNet

While this used to be a terribly difficult procedure at NPS, we fixed the difficulties, so now the process seems to work fine. Compiling must be performed on a machine running the Solaris operating system. There are two such machines available at NPS, `cincinnatus` and `virgo`. Both machines are running SunOS 5.5[1], and both machines are SPARCstation-20s. In order to compile SmartNet, perform the following tasks, in order. (This assumes you have already installed the code.)

---

[1]SunOS 5.5 is also called Solaris 2.5.

1. `telnet virgo` or `telnet cincinnatus`.

2. `cd  /SOLARIS`

3. `src/sn/configure --enable-use\_gnumake --enable-use\_gcc`

4. `make depend`

5. `make`

Other command line arguments to `configure` are listed below.

```
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]

Configuration:
  --cache-file=FILE       cache test results in FILE
  --help                  print this message
  --no-create             do not create output files
  --quiet, --silent       do not print `checking...´ messages
  --version               print the version of autoconf that
                           created configure

Directory and file names:
  --prefix=PREFIX         install architecture-independent
                           files in PREFIX [/usr/local]
  --exec-prefix=PREFIX    install architecture-dependent
                           files in PREFIX [same as prefix]
  --srcdir=DIR            find the sources in DIR
                           [configure dir or ..]
  --program-prefix=PREFIX prepend PREFIX to installed
                              program names
  --program-suffix=SUFFIX append SUFFIX to installed
                              program names
  --program-transform-name=PROGRAM run sed PROGRAM on
                              installed program names

Host type:
  --build=BUILD           configure for building on
                           BUILD [BUILD=HOST]
  --host=HOST             configure for HOST [guessed]
  --target=TARGET         configure for TARGET [TARGET=HOST]
```

```
Features and packages:
  --disable-FEATURE        do not include FEATURE
                             (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]   include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]     use PACKAGE [ARG=yes]
  --without-PACKAGE        do not use PACKAGE
                             (same as --with-PACKAGE=no)
  --x-includes=DIR         X include files are in DIR
  --x-libraries=DIR        X library files are in DIR
--enable and --with options recognized:
  --enable-use_gnumake     use the gnumake utility,
                             very nifty indeed
  --enable-use_gcc         use the gcc compiler instead of
                             native compiler
  --enable-use_DEBUG       make this thing DEBUG´ed
  --enable-use_OPTIMIZE    make this thing OPTIMIZE´ed
  --enable-use_RELEASE     make a releable version.
  --enable-use_static_link   make static linked binaries.
  --enable-use_purecov     make static linked binaries.
  --with-x                 use the X Window System
```

After several minutes, you will have compiled all the SmartNet binaries.

2.   USING THE SMARTNET SIMULATOR

This section assumes that the user has access to the SmartNet Users Guide [Ref. 10]. The Users Guide includes extensive instructions for and examples of commands for running SmartNet. The Users Guide does not include any information about running SmartNet in simulation mode, however. This section explains how to run SmartNet in simulation mode.

a.   Files

In order to run SmartNet in simulation mode, there is specific information that needs to be provided in certain files that will make SmartNet perform correctly.

i.   .smartnetrc

This file is required by SmartNet regardless of whether it is being run in simulation mode or not. The file may need to be altered, depending upon what we are trying to measure with the simulator. Here is a sample `.smartnetrc` file.

```
dbInFilename:    /users/work3/rkarmstr/SOLARIS/local/tests/hihi.0.0.dat
dbOutFilename:   /dev/null
scheduler:       OLB
rescheduleMode:  Off
debug:           none
debugFile:       /dev/null
verbosity:       v q
```

In the above `.smartnetrc` file, we would need to change the name of the input database file `dbInFilename` dependent upon the test we were running. Also, the scheduling algorithm used would need to be changed. Lastly, we may need to consider enabling the reschedule capability `rescheduleMode` in order to allow rescheduling to occur. The other lines can be altered as desired; explanation of all fields in the `.smartnetrc` file can be found in the Users Guide.

ii.   *Command File*

The command file lists jobs to be schedules and subsequently run by SmartNet. In simulation mode, SmartNet needs the command file data in order to know what jobs are to be scheduled and their execution simulated. An example of two types of command files is available in this appendix in Section 4. The command file can be anywhere in our directory structure; we will specify it by name and location when needed.

b.   Commands

In order to run SmartNet in simulation mode, several executables must be started in a particular order. First, the SmartNet-master must be started in simulation mode. This starts the SmartNet server in simulation mode as well as the SmartNet-queue. It also reads the SmartNet database for use by the scheduler. An example

156

database is located in Section 5 of this appendix. These programs basically start SmartNet. Next, we need to start the SmartNet logger, which enables logging of all job execution and scheduling messaging. After the SmartNet logger, be start the SmartNet submit program in simulation mode, which submits jobs, via the command file, to SmartNet so that these jobs can be scheduled.

After these commands have been entered, SmartNet will build a schedule, simulate the execution of the schedule, and stop. SmartNet master, queue, and server will still be running until killed. SmartNet submit also remains running, and must be killed by process number. SmartNet master and the rest can be killed with the command `sn-control -- OFF`. Note that the SmartNet logger will halt itself after the schedule has executed. Section 6 of this appendix has a sample script used to run through a single iteration of the process described above. Section 6 includes the command line arguments needed to start all the executables discussed here.

c.   Scripts

In order to make multiple runs of the SmartNet in simulation mode, we found it most helpful to use scripts. In the previous section, we discussed one of the many scripts used to help run SmartNet in simulation mode over and over again without the need for human intervention at the beginning and end of each test of SmartNet. Scripts were used throughout this research to simplify all the work performed.

Section 6 also includes a script used to run a set of experiments using multiple command files and multiple databases. It basically walks through the directory structure set up to house the experiments and performs sequences of tests. Instead of waiting at the terminal to type the commands, they have been scripted.

Section 6 also includes the Perl scripts written to parse data from the log files that the SmartNet logger writes. These log files include scheduling information and runtime information. We parsed this data using the file `parselog.pl`. This Perl script extracts the important information from the log files and puts it into another file, specified in the script. This parsed information is then parsed and averaged

again with the Perl script `collect.pl`, also found in Section 6. This script reduces the parsed data to a manageable form. The output is less than a page, and represents the run-time duration information of 60 separate executions of SmartNet.

3.   RUNNING SMARTNET IN SIMULATION MODE

Previous to this section, we discussed the necessary components of getting SmartNet ready to run, scripts used, and files/commands needed. Here, we put it all together in a step-by-step format in an attempt to make the process easier to follow.

1. Unpack SmartNet source code.

2. Compile SmartNet source code into SmartNet binaries.

3. Determine the experiments you need to perform.

   - Establish the directory structure you need for your output to be easily identified as being produced by a certain database or command file. You will need a `.smartnetrc` file in every directory from which you will run SmartNet.

   - Build your command file(s).

   - Build you database(s).

   - Ensure your `.smartnetrc` file(s) are calling the correct database file and scheduling algorithm.

   - Edit the `parselog.pl` and `collect.pl` files, as necesary. Each directory that you are running SmartNet from should contain a copy of both of these files. They should be able to be executed.

4. Build your scripts specific to the command files you intend to test. You will want one of each type in each directory from which you are running SmartNet.

5. Build your scripts specific to running different sets of SmartNet scripts listed previously. This is the big, "start it off" script.

6. Run the "start it off" script and collect your output.

Figure 43 shows how we set up our directory structure, to include naming conventions and files included.
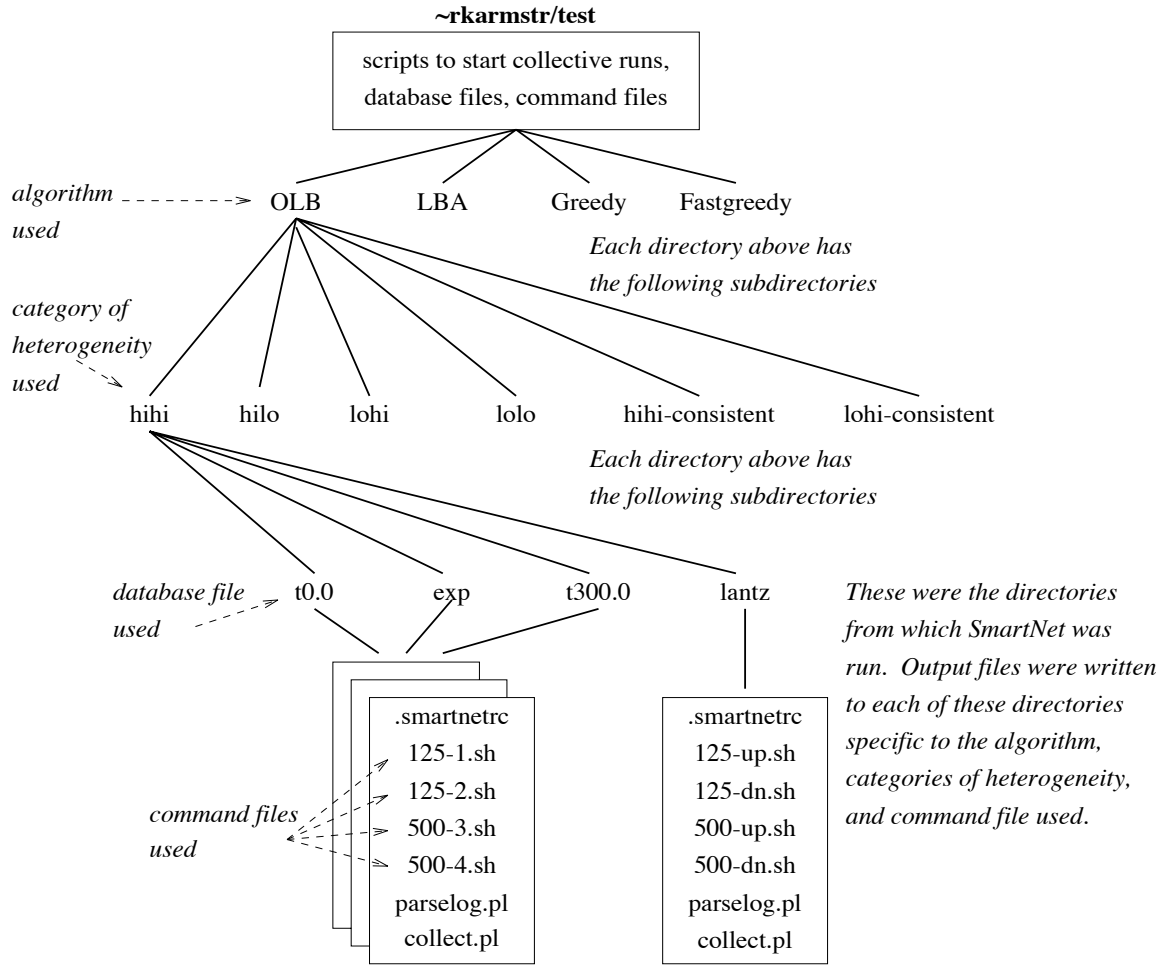
Figure 43. Directory Structure Used For Experiments. This was the directory structure we used throughout the conduct of this research.

4.   EXAMPLE COMMAND FILES

This section contains sample command files used in the conduct of this research.

a.   Command File — The Random Method

This sample command file is used to tell SmartNet the names of the jobs it needs to schedule. The jobs are read into SmartNet one at a time and with uniform randomness — hence, the name The Random Method.

```
model = job1
commandline = job1
cchars = 100
```

```
stdout = /dev/null
submit = 1

model = job4
commandline = job1
cchars = 100
stdout = /dev/null
submit = 1

model = job4
commandline = job1
cchars = 100
stdout = /dev/null
submit = 1

model = job3
commandline = job1
cchars = 100
stdout = /dev/null
submit = 1

model = job2
commandline = job1
cchars = 100
stdout = /dev/null
submit = 1

model = job2
commandline = job1
cchars = 100
stdout = /dev/null
submit = 1

model = job4
commandline = job1
cchars = 100
stdout = /dev/null
submit = 1

model = job3
commandline = job1
cchars = 100
```

```
stdout = /dev/null
submit = 1
```

b.   Command File — The Grouped Method

This sample command file also tells SmartNet which jobs it needs to schedule. It does so by grouping jobs. Note that `job1` is requested to run 25 times — hence, the grouped method.

```
model = job1
commandline = job1
cchars = 100
stdout = /dev/null
submit = 25

model = job2
commandline = job1
cchars = 100
stdout = /dev/null
submit = 25

model = job3
commandline = job1
cchars = 100
stdout = /dev/null
submit = 25

model = job4
commandline = job1
cchars = 100
stdout = /dev/null
submit = 25

model = job5
commandline = job1
cchars = 100
stdout = /dev/null
submit = 25
```

5.   EXAMPLE DATABASE FILE

```
//
// Armstrong sample database file
// for testing the SmartNetsimulator
//


//
// The number of Site objects
//
0


//
// The number of Machine objects
//
4


// The IP address is repeated for all machines because
// SmartNet tries to connect to the machine even in
// simulation mode, even though it will not run anything
// on the machine.  I gave it the IP address of hetero.
//
// Also, the names of te machines and jobs is notional
// See the SmartNet Users Guide for a more realistic
// database example.

machine1          // Machine name
Sun               // Architecture
131.120.2.1        // IP Address
Sun/Sparc 900
Notional
1                 // Relative cost
1                 // Is the machine notional?
NULL              // Site Name

machine2          // Machine name
Sun               // Architecture
131.120.2.1        // IP Address
Sun/Sparc 900
Notional
1                 // Relative cost
1                 // Is the machine notional?
NULL              // Site Name
```

```
machine3          // Machine name
Sun               // Architecture
131.120.2.1        // IP Address
Sun/Sparc 900
Notional
1                 // Relative cost
1                 // Is the machine notional?
NULL              // Site Name

machine4          // Machine name
Sun               // Architecture
131.120.2.1        // IP Address
Sun/Sparc 900
Notional
1                 // Relative cost
1                 // Is the machine notional?
NULL              // Site Name


//
// The number of Model objects
//
3

job1          // Model name
Bob´s Test Application1
1                 // idempotent [0|1]
1                 // The number of description lines
time

job2          // Model name
Bob´s Test Application2
1                 // idempotent [0|1]
1                 // The number of description lines
time

job3          // Model name
Bob´s Test Application3
1                 // idempotent [0|1]
1                 // The number of description lines
time
```

```
//
// The number of ModelMachine objects
//
12

machine1          // Machine name
job1 // Model name
NULL              // Group Name
normal // distribution type
300034.00 // moment-1 CHANGE FOR EACH
900102.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 3000.34              // Theoretical compute function
$0 * 0            // Theoretical Network function
1                 // Theoretical data use function
NULL              // Theoretical floating-point function
                  // Compute Data:
0                 // The amount of Experiential data
0                 // The amount of normalized Experiential data
                  // Network Data:
0                 // The amount of Experiential data


machine1          // Machine name
job2 // Model name
NULL              // Group Name
normal // distribution type
25.0 // moment-1 CHANGE FOR EACH
75.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 0.25                 // Theoretical compute function
$0 * 0            // Theoretical Network function
1                 // Theoretical data use function
NULL              // Theoretical floating-point function
                  // Compute Data:
0                 // The amount of Experiential data
0                 // The amount of normalized Experiential data
                  // Network Data:
```

```
0                   // The amount of Experiential data


machine1            // Machine name
job3 // Model name
NULL                // Group Name
normal // distribution type
1078.0 // moment-1 CHANGE FOR EACH
3234.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 10.78                      // Theoretical compute function
$0 * 0              // Theoretical Network function
1                   // Theoretical data use function
NULL                // Theoretical floating-point function
                    // Compute Data:
0                   // The amount of Experiential data
0                   // The amount of normalized Experiential data
                    // Network Data:
0                   // The amount of Experiential data


machine2            // Machine name
job1 // Model name
NULL                // Group Name
normal // distribution type
11.0 // moment-1 CHANGE FOR EACH
33.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 0.11                       // Theoretical compute function
$0 * 0              // Theoretical Network function
1                   // Theoretical data use function
NULL                // Theoretical floating-point function
                    // Compute Data:
0                   // The amount of Experiential data
0                   // The amount of normalized Experiential data
                    // Network Data:
0                   // The amount of Experiential data


machine2            // Machine name
job2 // Model name
```

```
NULL                // Group Name
normal // distribution type
1003.0 // moment-1 CHANGE FOR EACH
3009.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 10.03              // Theoretical compute function
$0 * 0              // Theoretical Network function
1                   // Theoretical data use function
NULL                // Theoretical floating-point function
                    // Compute Data:
0                   // The amount of Experiential data
0                   // The amount of normalized Experiential data
                    // Network Data:
0                   // The amount of Experiential data


machine2            // Machine name
job3 // Model name
NULL                // Group Name
normal // distribution type
93.0 // moment-1 CHANGE FOR EACH
279.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 0.93               // Theoretical compute function
$0 * 0              // Theoretical Network function
1                   // Theoretical data use function
NULL                // Theoretical floating-point function
                    // Compute Data:
0                   // The amount of Experiential data
0                   // The amount of normalized Experiential data
                    // Network Data:
0                   // The amount of Experiential data


machine3            // Machine name
job1 // Model name
NULL                // Group Name
normal // distribution type
239.0 // moment-1 CHANGE FOR EACH
717.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 2.39               // Theoretical compute function
```

```
$0 * 0            // Theoretical Network function
1                 // Theoretical data use function
NULL              // Theoretical floating-point function
                  // Compute Data:
0                 // The amount of Experiential data
0                 // The amount of normalized Experiential data
                  // Network Data:
0                 // The amount of Experiential data


machine3          // Machine name
job2 // Model name
NULL              // Group Name
normal // distribution type
8619.0 // moment-1 CHANGE FOR EACH
25857.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 86.19                  // Theoretical compute function
$0 * 0            // Theoretical Network function
1                 // Theoretical data use function
NULL              // Theoretical floating-point function
                  // Compute Data:
0                 // The amount of Experiential data
0                 // The amount of normalized Experiential data
                  // Network Data:
0                 // The amount of Experiential data


machine3          // Machine name
job3 // Model name
NULL              // Group Name
normal // distribution type
1950.0 // moment-1 CHANGE FOR EACH
5850.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 19.5                   // Theoretical compute function
$0 * 0            // Theoretical Network function
1                 // Theoretical data use function
NULL              // Theoretical floating-point function
                  // Compute Data:
0                 // The amount of Experiential data
0                 // The amount of normalized Experiential data
```

```
                        // Network Data:
0                       // The amount of Experiential data


machine4        // Machine name
job1 // Model name
NULL            // Group Name
normal // distribution type
30097.0 // moment-1 CHANGE FOR EACH
90291.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 300.97             // Theoretical compute function
$0 * 0          // Theoretical Network function
1               // Theoretical data use function
NULL            // Theoretical floating-point function
                // Compute Data:
0               // The amount of Experiential data
0               // The amount of normalized Experiential data
                // Network Data:
0               // The amount of Experiential data


machine4        // Machine name
job2 // Model name
NULL            // Group Name
normal // distribution type
75.0 // moment-1 CHANGE FOR EACH
225.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 0.75               // Theoretical compute function
$0 * 0          // Theoretical Network function
1               // Theoretical data use function
NULL            // Theoretical floating-point function
                // Compute Data:
0               // The amount of Experiential data
0               // The amount of normalized Experiential data
                // Network Data:
0               // The amount of Experiential data


machine4         // Machine name
job3 // Model name
```

```
NULL                // Group Name
normal // distribution type
204001.0 // moment-1 CHANGE FOR EACH
612003.0 // moment-2 CHANGE FOR EACH
0.0 // moment-3
$0 * 2040.01                    // Theoretical compute function
$0 * 0             // Theoretical Network function
1                  // Theoretical data use function
NULL               // Theoretical floating-point function
                   // Compute Data:
0                  // The amount of Experiential data
0                  // The amount of normalized Experiential data
                   // Network Data:
0                  // The amount of Experiential data



//
// The SNData default Override object:
//
NULL //Model name
NULL //Machine name
ExecutionEquation NULL
DataUseEquation NULL
NetworkEquation NULL
ComputeWeight 1
NetworkWeight 1
TheoreticalExecutionWeight 0.5
ExperientialExecutionWeight 0.5
OverrideExecutionWeight 0.5
TheoreticalNetworkWeight 0.5
ExperientialNetworkWeight 0.5
OverrideNetworkWeight 0.5
End_Override

//
// inter-site network information (bandwidth & latency)
//
End_NetMatrix
```

6.   EXAMPLE SCRIPTS
a.   Script for Starting and Running SmartNet: 125-1.sh

This is a script which makes it very easy to start and run SmartNet in simulation mode. The script will start SmartNet, execute a schedule in simulation mode, and then stop SmartNet. If you need to do this repetitively, the script should include multiple sequences of the below commands. We built scripts like the one below for each separate command file. They were located in the directory from which we ran SmartNet for that particular experiment.

```
#!/bin/ksh

# Start the master/server/queue
# -S is for simulation mode
# -s denotes the scheduling algorithm we desire to use.
#     This can also be spcified in the .smartnetrc file.
# -f denotes the name of the database file to be loaded
#     into SmartNet
smartnet-master -S -s OLB -f /users/work3/rkarmstr/tests/hihi.0.0.dat &

# This allows things to start up correctly
sleep 10

# Start the logger
# -n tells the logger how many jobs will be scheduled so that it
#     knows when to die
sn-log -n 125 -o test125-1-1.log &

# This allows things to start up correctly
sleep 3

# Start SmartNet submit
# -S is for simulation mode
# the required argument is the name of the command file
#  listing the jobs requests
sn-submit -S /users/work3/rkarmstr/tests/test125-1.cmd &

# Wait for the SmartNet logger to die
wait %2
```

```
# Kill SmartNet submit
kill -QUIT %3

# Wait for smartNet submit to die
sleep 10

# Kill the SmartNet master/server/queue
sn-control -- OFF
```

b. Script for Running Experiments: tt0.0.sh

```
#! /bin/ksh

# This is a script to run all 0.0 variance tests
# for hihi|hilo|lohi|lolo|linear heterogeneous sets
# on olb|lba|greedy|fastgreedy algorithms.

# olb tests
mail rkarmstr < /users/work3/rkarmstr/SOLARIS/local/tests/mmolb
cd /users/work3/rkarmstr/SOLARIS/local/tests/olb/hihi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/olb/hilo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/olb/lohi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
```

```
cd /users/work3/rkarmstr/SOLARIS/local/tests/olb/lolo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/olb/linear/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl

# lba tests
mail rkarmstr < /users/work3/rkarmstr/SOLARIS/local/tests/mmlba
cd /users/work3/rkarmstr/SOLARIS/local/tests/lba/hihi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/lba/hilo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
```

```
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/lba/lohi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/lba/lolo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/lba/linear/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl

# greedy tests
mail rkarmstr < /users/work3/rkarmstr/SOLARIS/local/tests/mmgreedy
cd /users/work3/rkarmstr/SOLARIS/local/tests/greedy/hihi/t0.0
```

```
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/greedy/hilo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/greedy/lohi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/greedy/lolo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
```

```
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/greedy/linear/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl

# fastgreedy tests
mail rkarmstr < /users/work3/rkarmstr/SOLARIS/local/tests/mmfastgreedy
cd /users/work3/rkarmstr/SOLARIS/local/tests/fastgreedy/hihi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/fastgreedy/hilo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/fastgreedy/lohi/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
```

```
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/fastgreedy/lolo/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
cd /users/work3/rkarmstr/SOLARIS/local/tests/fastgreedy/linear/t0.0
125-1.sh
sleep 10
125-2.sh
sleep 10
500-3.sh
sleep 10
500-4.sh
sleep 30
parselog.pl
collect.pl
mail rkarmstr < /users/work3/rkarmstr/SOLARIS/local/tests/mmdone
```

7.   EXAMPLE PARSE SCRIPTS
a.   Parsing Run-Time Data From Log Files: parselog.pl

```perl
#!/bin/perl

# This Perl script is meant to run on version Perl 5.0.
# Perl 5 is loaded onto virgo.
# This script is written for 0 variance tests.  That is why
# it only looks for 15 repetitions of the logfile.  For tests
# where you run SmartNet more than once for each command file,
# you need to change the "1" to "15" or whatever number
# of reps you run.  See the note at each place needing change.

use Cwd;

while (<*.log>) {
    chmod 0600, $_;
}
@files = ("test125-1", "test125-2", "test500-3", "test500-4");

for ($yy = 0; $yy < 4; $yy++) {
    open(OUT, ">parse-@files[$yy].log");
    print OUT "Data parsed from file:\t@files[$yy].log\n\n\n";

    $dir = cwd();
    print OUT "Output from directory:\n\t$dir\n\n";
    $sum = 0;
    for ($ix = 0; $ix < 1; $ix++) {  ##Need to change the "1" to "15" normally
$iy = $ix + 1;
$aa = 0;
$flag = 0;
$count = 0;
$machine1 = 0;
$machine2 = 0;
$machine3 = 0;
$machine4 = 0;
$machine5 = 0;
$machine6 = 0;
$machine7 = 0;
$machine8 = 0;
$machine9 = 0;
$machine10 = 0;
$job1 = 0;
```

```perl
$job2 = 0;
$job3 = 0;
$job4 = 0;
$job5 = 0;

open(IN, "@files[$yy]-$iy.log") or die "Can´t open @files[$yy]-$iy.log\n";
while ($line = <IN>) {
    ($one, $two, $three, $four, $five, $six) = split(" ", $line);
    if (($one eq "SCHED") && ($flag == 0) ) {
if($four eq "host<machine1>") {
    $machine1++;
}elsif ($four eq "host<machine2>") {
    $machine2++;
}elsif ($four eq "host<machine3>") {
    $machine3++;
}elsif ($four eq "host<machine4>") {
    $machine4++;
}elsif ($four eq "host<machine5>") {
    $machine5++;
}elsif ($four eq "host<machine6>") {
    $machine6++;
}elsif ($four eq "host<machine7>") {
    $machine7++;
}elsif ($four eq "host<machine8>") {
    $machine8++;
}elsif ($four eq "host<machine9>") {
    $machine9++;
}elsif ($four eq "host<machine10>") {
    $machine10++;
}
    }
    if (($one eq "SCHED") && ($flag == 0) ) {
if($five eq "model<job1>") {
    $job1++;
}elsif ($five eq "model<job2>") {
    $job2++;
}elsif ($five eq "model<job3>") {
    $job3++;
}elsif ($five eq "model<job4>") {
    $job4++;
}elsif ($five eq "model<job5>") {
    $job5++;
```

```perl
}
    }
    if (($one eq "START") && ($flag == 0) ) {
$three =~ s/time<//g;
$three =~ s/>//g;
$start[$ix] = $three;
print OUT "Run $iy: start:\t\t$start[$ix]\n";
$flag = 1;
    }
    if ($one eq "DONE") {
$count++;
if(($count == 125) and ($yy < 2)) {
    $three =~ s/time<//g;
    $three =~ s/>//g;
    $end[$ix] = $three;
    print OUT "Run $iy:   end:\t\t$end[$ix]\n";
}elsif(($count == 500) and ($yy > 1)) {
    $three =~ s/time<//g;
    $three =~ s/>//g;
    $end[$ix] = $three;
    print OUT "Run $iy:   end:\t\t$end[$ix]\n";
}
    }
}
$duration[$ix] = $end[$ix] - $start[$ix];
print OUT "DURATION for Run $iy is:  $duration[$ix]\n\n";
$sum = $sum + @duration[$ix];
close IN;
print OUT "Number of machine1 assignments:   $machine1\n";
print OUT "Number of machine2 assignments:   $machine2\n";
print OUT "Number of machine3 assignments:   $machine3\n";
print OUT "Number of machine4 assignments:   $machine4\n";
print OUT "Number of machine5 assignments:   $machine5\n";
print OUT "Number of machine6 assignments:   $machine6\n";
print OUT "Number of machine7 assignments:   $machine7\n";
print OUT "Number of machine8 assignments:   $machine8\n";
print OUT "Number of machine9 assignments:   $machine9\n";
print OUT "Number of machine10 assignments:  $machine10\n\n";

print OUT "Number of job1 assignments:      $job1\n";
print OUT "Number of job2 assignments:      $job2\n";
print OUT "Number of job3 assignments:      $job3\n";
```

```perl
print OUT "Number of job4 assignments:     $job4\n";
print OUT "Number of job5 assignments:     $job5\n\n";
    }


    $average = $sum/1;  ## Need to change to "15" normally
    print OUT "\nAverage runtime for @files[$yy] is:  $average\n";
    close OUT;
}
```

b.   Collecting Run-Time Data

```perl
#! /bin/perl

use Cwd;

@files = <parse-*.log>;

$dir = cwd();
($first, $users, $work3, $rkarmstr, $solaris, $here, $tests, $algorithm, $heteroge

$ix = 0;
open(OUT, ">$algorithm.collect") or die "Cannot open $algorithm.collect\n";

print OUT "Algorithm:\t$algorithm\nHeterogeneity:\t$heterogeneity\nTest run:\t$var

while (@files[$ix]){
    open(IN, (shift @files)) or die "Can't open (shift @files)\n";
        while (<IN>) {
    if(/Average runtime for test([0-9.]+)-([0-9]) is:  *([0-9.]+)/) {
$average = $3;
print OUT "The average runtime for test$1-$2 is:  $average \n";
    }
    }



    close(IN);

}
close(OUT);
```

LIST OF REFERENCES

[1] Richard Freund, Debbie Hensgen, Taylor Kidd, and Lantz Moore. Smartnet: A Scheduling Framework for Heterogeneous Computing. *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, 1996.

[2] Lantz Moore. System Software Developers Desperately Need Better Simulation Tools. In Jeffrey W. Wallace, Terrence G. Beaumariage, and Yasser Dessouky, editors, *Object-Oriented Simulation Conference (OOS '97)*. The Society for Computer Simulation International, 1997.

[3] Richard Freund, Taylor Kidd, and Debra Hensgen. Performance Analysis and Measurement in SmartNet. Prepared by the SmartNet Heterogeneous Computing Team on 961213.

[4] Stephen L. Ambrosius, Stephen L. Scott, Richard F. Freund, and Howard Jay Siegel. Work-based Performance Measurement and Analysis of Virtual Heterogeneous Machines. *Heterogeneous Computing Workshop*, 1996.

[5] Taylor Kidd, Debra Hensgen, Richard Freund, Matt Kussow, and Mark Campbell. Compute Characteristics: A Useful Characterization of Job Runtimes. In preparation for submission (1997).

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[7] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1996.

[8] Cray Research, Inc. *CRAY Y-MP EL Functional Description*, 1992.

[9] Jesse C. Benton and Michael J. Lemanski. Simulation for SmartNet Scheduling of Asynchronous Transfer Mode Virtual Channels. Master's thesis, U.S. Naval Postgraduate School, June 1997.

[10] Naval Command, Control, and Ocean Surveillance Center, Research, Development, Test and Evaluation Division, Code 422, 53140 Gatchell Road, San Diego, CA 92152-7400. *SmartNet Scheduling Tool v2.6 Users Guide*, June 1996.

[11] Hartmut Pohlheim. Genetic and Evolutionary Algorithm Toolbox for use with Matlab (GEATbx). WWW: http://www.systemtechnik.tu-ilmenau.de/ pohlheim/GA_Toolbox/algoverv.html.

[12]Paul Coddington. Simulated Annealing and Optimization. WWW: http://www.npac.syr.edu/users/gcf/cps713montecarlo/node133.html. Northeast Parallel Architectures Center at Syracuse University.

[13]Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis, Second Edition*. McGraw-Hill, Inc., New York, 1991.

[14]Sheldon M. Ross. *A Course in Simulation*. Macmillan Publishing Company, New York, 1990.

[15]Donald E. Knuth. *The Art of Computer Programming*, volume 2, Seminumerical Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1981.

[16]Mathrubootham Janakiraman. Simulation Results for Heuristic Algorithms for Scheduling Precedence-Related Tasks in Heterogeneous Environments. Master's thesis, University of Cincinnati, 1996.

[17]Sun Microsystems. *SunOS Reference Manual, Volume I*. Revision A of 27 March 1990.

[18]David Bailey et al. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[19]Peter Pacheco. A User's Guide to MPI. Technical report, Department of Mathematics, University of San Fransisco, March 1995.

[20]A. Beguelin et al. *HeNCE: A User' Guide*. Oak Ridge National Laboratory and University of Tennessee, December 1992. The document itself is available on the web at cs.utk.edu.

INITIAL DISTRIBUTION LIST

1.Defense Technical Information Center   2
    8725 John J. Kingman Road., Ste 0944
    Ft. Belvoir, VA 22060-6218

2.Dudley Knox Library   2
    Naval Postgraduate School
    411 Dyer Rd.
    Monterey, CA 93943-5101

3.Director, Training and Education   1
    MCCDC, Code C46
    1019 Elliot Road
    Quantico, VA 22134-5027

4.Director, Marine Corps Research Center   2
    MCCDC, Code C40RC
    2040 Broadway Street
    Quantico, VA 22134-5107

5.Director, Studies and Analysis Division   1
    MCCDC, Code C45
    3300 Russell Road
    Quantico, VA 22134-5130

6.Marine Corps Representative   1
    Naval Postgraduate School
    Code 037, Bldg. 234, HA-220
    699 Dyer Road
    Monterey, CA 93940

7.Marine Corps Tactical Systems Support Activity   1
    Technical Advisory Branch
    Attn: Maj. J.C. Cumiskey
    Box 555171
    Camp Pendleton, CA 92055-5080

8.Debra Hensgen   5
    Naval Postgraduate School
    Code CS/Hd, Computer Sciences Dept.
    833 Dyer Rd.
    Monterey, CA 93943-5118

9. John Falby   1
   Naval Postgraduate School
   Code CS/Fa, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

10. H.J. Siegel   1
    Purdue University
    Room 325, EE Building
    School of Electrical and Computer Engineering
    1285 Electrical Engineer Building
    West Lafayette, IN 47907-1285

11. Richard Freund, Chief Scientist   1
    Heterogeneous Computing Team
    NCCOSC RDTE Div 4221 Rm 341A
    53118 Gatchell Road
    San Diego, CA 92152-7446

12. Taylor Kidd   1
    Naval Postgraduate School
    Code CS/Kt, Computer Sciences Dept.
    833 Dyer Rd.
    Monterey, CA 93943-5118

13. Viktor Prasanna   1
    University of Southern California
    Department of EE-Systems, EEB 200C
    3740 McClintock Ave.
    Los Angeles, CA 90089-2562

14. Major Bob Armstrong, USMC   1
    5775 Hall Lane
    Twentynine Palms, CA 92277-2195

15. Mr. and Mrs. R. K. Armstrong   1
    140 Haverford Drive
    Nashville, TN 37205